METAHEURÍSTICAS

PRÁCTICA 3.A: TÉCNICAS DE BÚSQUEDA BASADAS EN TRAYECTORIAS 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 3 – TÉCNICAS DE BÚSQUEDA BASADAS EN TRAYECTORIAS 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 Trayectorias,* en concreto, en la resolución del problema de la máxima diversidad (MDP) descrito en el Seminario 4.

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 (ES, BMB, ILS, ILS-ES), 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 Enfriamiento Simulado

Primeramente, trataremos brevemente de explicar en qué consiste un algoritmo basado en enfriamiento simulado, para posteriormente especificar en profundidad mediante el análisis del pseudocódigo y la inclusión generada de la hibridación con otros algoritmos. Estos se caracterizan, además de por ser algoritmos de búsqueda metaheurísticos (que tratan de encontrar una buena aproximación del coste al evaluar la función objetivo), por ser una adaptación del algoritmo de Metrópolis (estudiado en clase), utilizado para generar muestras de estados de un sistema termodinámico.

En primer lugar, generaremos una solución totalmente aleatoria y factible que utilizaremos para calcular la temperatura inicial de nuestro sistema a través de la que, mediante el enfriamiento, iremos moviéndonos hacia soluciones mejores o peores atendiendo a una serie de criterios. Para ello, dispondremos de un modelo tanto para calcular la temperatura inicial del sistema como para la final, quedando como:

$$\boldsymbol{T_0 = \frac{\mu \cdot C(S_0)}{-\ln(\phi)}} \qquad \boldsymbol{T_f = 10^{-3}}$$

donde:

- μ y ϕ son constantes con un valor 0,3.
- $C(S_0)$ equivale al coste de la solución inicial aleatoria generada.

Una vez obtenida la temperatura inicial y final, realizaremos la aplicación de un tipo de enfriamiento clásico (en vista de que con el esquema de Cauchy modificado no pude obtener resultados satisfactorios), que se basa en el enfriamiento de la temperatura mediante el producto de una constante de enfriamiento $0{,}9 \leq \boldsymbol{\alpha} < 1$ de manera que nuestra nueva temperatura vendrá definida por $\boldsymbol{T} = \alpha \cdot T$, produciéndose así un enfriamiento controlado que sea capaz de producir una convergencia suficiente como para obtener resultados bastante satisfactorios antes de alcanzar la temperatura final de enfriamiento (T_f).

A continuación, calcularemos aquellas constantes que nos servirán de utilidad a la hora de saber cuándo detener nuestro algoritmo y bajo qué condiciones debe detenerse.

Por una parte, necesitaremos saber cuántos vecinos querremos generar antes de comenzar a enfriar la temperatura, siendo este $\boldsymbol{maxVecinos = 10 \cdot n}$, donde n es el número total de elementos de nuestra matriz de distancias totales; al igual que necesitaremos saber tras cuántos éxitos realizaremos un enfriamiento (siempre que aún no hayamos generado todos los vecinos de la evaluación actual), que vendrá definida por $\boldsymbol{maxExitos = 0{,}1 \cdot maxVecinos}$**.** Estas variables serán las encargadas de condicionar la salida del bucle interno de nuestro algoritmo de enfriamiento, ya que, si hemos alcanzado **maxVecinos** ó **maxExitos,** finalizaremos la ejecución de la evaluación actual del bucle interno para realizar un enfriamiento volviendo al bucle interno. Esto continuará siempre que no alcancemos la temperatura final, o no hagamos un número máximo de enfriamientos definidos por ***maxEnfriamientos*** definido por el cociente entre el máximo número de iteraciones (100000) y **maxVecinos,** ó bien porque no hemos obtenido éxito alguno en la evaluación interna tras la generación de todo el **vecindario actual.**

En lo referente a la **generación del vecindario**, lo haremos a través del uso de una función ***intAleatorio*** que recibirá la lista solución actual para realizar un intercambio aleatorio entre un **i** perteneciente a la lista solución, y un **j** perteneciente a la lista de candidatos a solución, de manera que la solución obtenida seguirá siendo factible. Tras obtener nuestro vecino, calcularemos la diferencia de costes mediante la función objetivo para decidir entre el aporte generado por la solución inicial o la generada por el vecino (entre otras condiciones, encargadas de permitir que se pueda entrar en soluciones peores, pudiendo escapar de óptimos locales).

Representación de la solución:

Utilizaremos un tipo de representación real, mediante una lista de tamaño m entre los n elementos de la forma Sel = (x1, … , x_m) donde cada x corresponde a un elemento concreto dentro de la matriz de distancias representando su posición.

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 nuestra lista, sea la máxima posible, atendiendo a que la sumatoria de elementos seleccionados en nuestra solución sea exactamente m, que variará en función de los datos del problema (50, 200…).

Ciclo de enfriamiento:

```
Inicio
      1. Inicializar listaSol <- generación aleatoria
      2. Calcular T_inicial
      3. Inicializar mejorSol <- listaSol
      4. Inicializar T_actual <- T_inicial
      Repetir
            Mientras (no lleguemos al límite de generaciones del vecindario ó
                      al límite de éxitos)
                  4. Vecino <- Int(listaSol, n) /* Intercambio simple
                  5. Diferencia <- coste(Vecino) - coste(listaSol)
                  Si ((diferencia > 0) or (U(0,1) ≥ exp(-diferencia / T_actual)) entonces
                        6. listaSol <- Vecino
                        Si (Coste(listaSol) > Coste(mejorSol)
                              7. MejorSol <- listaSol
                        Endif
                  Endif
                  8. T_actual <- α · T_actual  /* α = 0,95
            Endwhile
      Mientras (T_actual ≤ T_final y no lleguemos al límite de enfriamientos)
      9. return(mejorSol)
Fin
```

donde:

- listaSol es una lista solución inicial generada aleatoriamente de tamaño m entre los n elementos.
- Int(listaSol, n) es una función que realiza un intercambio entre las soluciones de listaSol y el resto de números hasta n que no se encuentran en listaSol, así obtenemos una nueva solución con una única mutación.
- Coste(listaSol) es la función objetivo encargada de evaluar la solución actual devolviendo un valor de coma flotante. Función calculoCosteFinal(listaSol, vecDistancias).

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 mutación:

```
IntAleatorio (listaSol, n)
       1. Creamos un numero aleatorio solRand <- Rand(0,n)
       2. Creamos otro número aleatorio candRand <- Rand(0,n)
       3. Creamos vecino <- lista auxiliar
       Mientras solRand ∉ listaSol ó candRand ∈ listaSol
              4. Generar nuevamente solRand
              5. Generar nuevamente candRand
       Endwhile
       6. Avanzar en listaSol hasta posición de solRand
       7. listaSol[solRand] <- candRand
       8. vecino <- listaSol
       9. return vecino
Fin
```

Enfoque Búsqueda Multiarranque Básica (BMB)

A continuación, trataré de realizar una breve explicación, al igual que con el anterior algoritmo sobre cómo deberíamos de implementar este segundo algoritmo basado en trayectorias conocido como la *Búsqueda Multiarranque Básica.*

Este algoritmo consiste básicamente en generar un número determinado de soluciones aleatorias iniciales, a partir de las que trataremos de buscar la máxima optimización mediante el uso del algoritmo de ***Búsqueda Local*** desarrollado durante la práctica 1 de la asignatura (junto al algoritmo Greedy).

Únicamente tendremos que ejecutar una vez nuestro programa sobre cada caso de los mencionados en la introducción de esta práctica, realizando un total de 10 iteraciones, es decir, aplicando BL sobre cada una de las soluciones aleatorias generadas inicialmente, sobre un máximo de 10000 evaluaciones por cada ejecución de nuestro algoritmo de BL. Una vez aplicadas las búsquedas locales convenientes, devolveremos la mejor solución encontrada durante el proceso.

Esquema BMB:

```
Inicio
       10. MatrizSol <- Generación 10 soluciones aleatorias
           /* listaSol_i
       11. MatrizCand <- Generar 1 lista por cada solución con
           los elementos no seleccionados de esta. /* listaCand
           Mientras aún queden listas a las que aplicar BL en
           matrizSol
              3. listaSol_i <- BL (vecDistancias, LIMITE,
              listaSol, listaCand)
              4. Escogemos siguiente listaSol_i
           Endwhile
Fin
```

Función barajar:

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

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 ¡Mejora
                    17. comprobar siguiente elemento en Sel
                    18. iterador <- iterador++
              endif

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

Enfoque Búsqueda Local Reiterada (ILS)

El tercer algoritmo a implementar dentro de la tercera práctica de la asignatura es conocido como ***Iterative Local Search***, que consiste principalmente en la generación aleatoria de una solución inicial al que le aplicaremos el algoritmo de BL, mencionado anteriormente tanto en otras prácticas como en apartados de esta misma práctica, con el fin de obtener una primera solución optimizada.

Tras la obtención de esta primera mejor solución (junto al respectivo coste), comenzaremos a iterar a través de un bucle que nos permitirá realizar un máximo de **10 aplicaciones** del algoritmo de BL, teniendo en cuenta la aplicación de dicho algoritmo para la primera solución inicial generada que se trata de optimizar.

Durante cada ciclo de nuestro bucle interno, aplicaremos a la que actualmente es nuestra mejor solución (en principio la inicial), un número de mutaciones (a través de la función ***intAleatorio***) sobre esta igual a **intercambios = 0,1 · m**, donde m es el número de elementos pertenecientes a la lista solución.

Finalmente, y antes de evaluar la lista actual mediante la función objetivo, aplicaremos a dicha lista "mutada" **una búsqueda local** encargada de optimizar el coste producido por esta lo más rápido posible dando pie a una evaluación de la lista optimizada que "luchará" con la lista que mejor solución haya aportado hasta el momento, sustituyendo tanto la lista como el coste en caso de salir victoriosa esta nueva lista obtenida durante la iteración.

Esquema ILS:

```
Inicio
      5. ListaSol <- Inicializar solución.
      6. ListaCand <- Inicializar candidatos no seleccionados.
      7. CosteMejor <- calculoCosteFinal(listaSol, vecDistancias)
      8. Aplicar BL(vecDistancias, LIMITE, listaSol, listaCand
      9. ListaMejor <- listaSol
      Mientras(no realicemos todas las BL convenientes)
            4. Vecino <- intAleatorio(listaSol, n) /* Realizamos 0,1*m
            5. Aplicamos BL sobre Vecino
            6. costeVecino <- calculoCosteFinal(vecino, vecDistancias)
            Si costeMejor < costeVecino
                  7. costeMejor = costeVecino
                  8. listaMejor <- vecino
            sino
                  9. listaMejor <- vecino
            Endif
            10. NumBL++;
      Endwhile
Fin
```

Inicialización de la solución:

```
Inicio (Sol, m)
       Desde 0
       2. 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
```

Hibridación ILS - ES

La última tarea que se nos encomendó fue la de realizar una hibridación de los algoritmos ILS y ES que, en lugar de aplicar la BL sobre todas nuestras soluciones que fuéramos obteniendo, aplicaríamos el esquema del algoritmo de Enfriamiento Simulado con el fin de obtener resultados y tiempos diferentes para poder analizar el impacto de cada uno de los algoritmos para el problema elegido.

Finalmente, tras la implementación y prueba de este último algoritmo, pude encontrar diferencias poco significantes en cuanto a la desviación respecto al resultado óptimo, aunque no podría decir lo mismo del tiempo obtenido, ya que en comparación al ES, obtuve tiempos de más del doble de tiempo.

Esquema ILS-ES:

```
Inicio
       1. ListaSol <- Inicializar solución.
       2. CosteMejor <- calculoCosteFinal(listaSol, vecDistancias)
       3. Aplicar ES(listaSol, vecDistancias, costeMejor, n)
       4. ListaMejor <- listaSol
       Mientras(no realicemos todas las BL convenientes)
              4. Vecino <- intAleatorio(listaSol, n) /* Realizamos 0,1*m
              5. costeVecino <- calculoCosteFinal(vecino, vecDistancias)
              6. Aplicamos ES sobre Vecino
              Si costeMejor < costeVecino
                     7. costeMejor = costeVecino
                     8. listaMejor <- vecino
              sino
                     9. listaMejor <- vecino
              Endif
              10. NumBL++;
       Endwhile
Fin
```

Comparación entre enfoques

Algoritmo	Desviación	Tiempo (s)
Greedy	1,37	203,33
BL	1,04	3,10
ES	0,99	32,36
BMB	0,45	0,81
ILS	0,58	0,65
ILS-ES	0,66	119

Una vez realizadas todas las pruebas convenientes sobre los casos de problemas facilitados por los profesores de la asignatura, podemos apreciar algunas diferencias bastante significativas tanto en cuestión de tiempos, como de optimalidad.

En primer lugar, me gustaría hablar un poco sobre el algoritmo de Enfriamiento Simulado, ya que considero que este algoritmo, a diferencia de los demás, es bastante dependiente de los ajustes que se realicen en función del problema y las condiciones, ya que serán estas las que decidan si el algoritmo ajusta mejor, pero, más rápidamente o más lentamente. En un primer golpe de suerte, pude implementarlo correctamente gracias al esquema de enfriamiento clásico tomando valores de 0.95 (como algunos compañeros me recomendaron) para alpha y 0,3 tanto para mu como para fi (como los profesores recomendaban en el guión), por lo que realmente no me supuso una gran complicación el hecho de poder conseguir resultados suficientemente satisfactorios, aunque ciertamente bastante lentos en comparación con los demás algoritmos desarrollados durante esta práctica. Al final imagino que este tipo de algoritmo (al igual que los genéticos desarrollados en la práctica anterior) dependerán notoriamente del problema que estemos tratando y de los tiempos de los que dispongamos, ya que como con los algoritmos genéticos, se pueden obtener resultados muy satisfactorios en determinadas situaciones o campos.

Por otra parte, me sorprendió la facilidad de la implementación de la búsqueda Multiarranque básica acompañada, además de unos resultados, hasta el momento, de lo más cercanos al óptimo, lo que me hizo pensar en que el factor de aleatoriedad es capaz de jugar un papel tan importante en un algoritmo ligado tanto a la convergencia de resultados óptimos como a la velocidad de obtención de dichos resultados. He de decir que este ha sido el algoritmo que más me ha sorprendido por su ínfima dificultad y sus óptimos resultados, por lo que creo que, para casi todos los problemas, siempre podríamos probar en una primera instancia, lanzar este tipo de algoritmo enfocado al problema que estemos tratando, con el fin de conseguir unos primeros resultados acerca del problema que tratamos, que seguramente serán suficientemente buenos en un tiempo más que razonable.

En cuanto al tercer algoritmo desarrollado durante esta práctica (Algoritmo Búsqueda Local Reiterada), pude comprobar como los resultados obtenidos eran algo peores, lo que pensé que estaba ligado a la serie de intercambios realizados sobre una misma lista, que además de permitirte escapar de óptimos locales, puede obtener resultados bastante peores. Puede resultar un poco contradictorio el pensar que, al realizar el intercambio aleatorio en este algoritmo, estamos aplicando una aleatoriedad similar a la realizada en el BMB, aunque profundizando un poco en ambos algoritmos podremos denotar que no, me explico; en la búsqueda Multiarranque, nuestro objetivo era el partir de diferentes soluciones iniciales generadas, por lo que cada par de ellas pueden parecerse en algo, o, por el

contrario, ser absolutamente diferentes. Este hecho aplica una mayor diversidad a la hora de encontrar soluciones tras la aplicación de la BL. Por otro lado, en el algoritmo ILS, la diferencia que encontramos es que, a partir de una primera solución generada aleatoriamente, nuestro vecindario dependerá única y exclusivamente de los intercambios que se realicen sobre esta primera solución inicial obtenida, por lo que la diversidad en comparación, tiene sentido que sea mucho menor para el ILS que para la BMB.

De todas formas, al igual que podemos ver como BMB obtiene mejores resultados, podemos comprobar cómo consume mayores tiempos en promedio que ILS, debido a la aplicación de 10 BL sobre 10 diferentes soluciones aleatorias generadas anteriormente (en BMB); en comparación con los 5 intercambios aleatorios realizados en ILS. Es por ello, que, si tuviera que decantarme por alguno de los algoritmos, debería de atender siempre tanto al problema que estemos tratando, como al espacio del problema, y sobre todo, al hecho de querer priorizar en optimalidad o en tiempo de ejecución, aunque por lo general me decantaría por el BMB.

Por último, en referencia a la hibridación de ILS con ES, cabe destacar que los resultados obtenidos (quiero creer que por el tipo de problema y la hibridación) son significativamente mejores que los de la ES implementada en solitario, ya que creo que al introducir un esquema de enfriamiento en el algoritmo ILS, nos permite evitar un poco el problema que teníamos anteriormente en ILS donde una vez generada la solución inicial, realizaríamos una serie de intercambios aleatorios como mecanismo de diversidad. Ahora, además de tener este mecanismo original del ILS, encontramos otro como es el esquema de enfriamiento que nos permite incluir un poco más de diversidad a nuestro problema en comparación con el ES original, aunque en términos de tiempo, también era de esperar que utilizar el esquema de enfriamiento siempre es un poco más laborioso que los demás, por lo que entre eso y la aplicación del ILS, creo que es razonable el porqué de los tiempos tan elevados. En cuanto a la relación Desviación-Tiempo, creo que la diferencia es bastante grande, por lo que tal vez trataría de realizar otro tipo de hibridación en función de nuestras necesidades de tiempo o coste (tal vez ILS-BMB ó BMB-ES, aunque esta diera tiempos similares a ILS-ES).

BIBLIOGRAFÍA:

- Guion de la práctica:
https://pradogrado1920.ugr.es/pluginfile.php/395032/mod_resource/content/8/Guion%20P3a%20Trayectorias%20MDP%20MHs%202019-20.pdf

- Seminario 3 de la asignatura:
https://pradogrado1920.ugr.es/pluginfile.php/395058/mod_resource/content/11/Sem04-Problemas-Trayectorias-MHs-19-20.pdf

- Tema 5:
https://pradogrado1920.ugr.es/pluginfile.php/394984/mod_resource/content/11/Tema05-Metodos%20basados%20en%20trayectorias-19-20.pdf

- Wikipedia
https://es.wikipedia.org/wiki/Algoritmo_de_recocido_simulado

- Páginas Externas
http://www.cs.us.es/~fsancho/?e=205