Metaheurísticas (2018-2019)
Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Práctica 2: Algoritmos Genéticos y Meméticos para el Problema de Máxima Diversidad (MDP)

Víctor Bricio Blázquez
vbricio@correo.ugr.es
Grupo 2: Miércoles de 17:30h a 19:30h

6 de mayo de 2019

Índice

1. Descripción del problema

El problema se trata de elegir los m elementos de entre una familia de n elementos que cumplan que la distancia entre estos m elementos es máxima.
Una de las partes más importantes de este enunciado es el concepto de distancia, pues se pueden tomar muchos tipos de distancia para atacar este problema. En este caso se nos pide que la distancia a tomar sea la suma de todas las distancias entre todos los elementos de nuestra solución (denominada MaxSum).
Llega ahora el concepto de solución, diremos que un conjunto de números es solución si cumple las siguientes características, para los algoritmos Greedy y Búsqueda Local:

- Contiene exactamente m elementos.
- No existen elementos repetidos.
- El orden de los elementos no es relevante.

En cambio, para los algoritmos genéticos y meméticos una solución cumple las siguientes características:

- Contiene n elementos (booleanos).
- Hay exactamente m unos (true).

2. Descripción de las estructuras de datos

Para afrontar este problema he utilizado un lector de datos que convierte los archivos que se nos proporcionan en una matriz triangular, dejando la diagonal y el otro triángulo de la matriz como ceros.
Para los algoritmos Greedy y Búsqueda Local, he desarrollado una serie de vectores que son de notable interés:

- **solucion**: En este vector se estará construyendo una solución o será en sí una solución.
- **usados**: En este vector estarán todos los elementos que se hayan probado para mejorar una solución y no la hayan mejorado.
- **mundo**: En este vector estarán todos los elementos que no estén en la solución o en usados.

La función objetivo (denominada score) suma las distancias que relacionan todos los elementos de la solución con todos los elementos de la solución. Tal y cómo lo tengo diseñado estaría sumando todo dos veces y estaría sumando la distancia entre un elemento y él mismo. Pero el hecho de que la diagonal y el otro triángulo de la matriz estén llenas de ceros soluciona estos problemas.
Sin embargo, para los algoritmos genéticos y meméticos, he diseñado las siguientes matrices y vectores de interés:

- **matrixSolucionesBin**: Consiste en una matriz que tiene el número de columnas igual al número de cromosomas que se tratan, y el número de filas igual a n. Aquí se guardan las soluciones actuales.
- **matrixDeSeleccion**: Consiste en una matriz de las dimensiones de la anterior en la que se guardan las soluciones una vez seleccionadas por el proceso de selección correspondiente.
- **matrixDeHijos**: Consiste en una matriz de las dimensiones de las anteriores en la que se guardan los hijos, de las soluciones almacenadas en la matriz anterior, tras el cruce que corresponda.
- **scoreTotalBin**: Se trata de un vector en el que se almacenan los scores de las soluciones almacenadas en la matriz de soluciones.

En este caso, la función objetivo calcula las distancias entre los elementos de una posición de la matriz de soluciones. Puesto que para desarrollar estos algoritmos hemos decidido usar la representación booleana de las soluciones, a este score se le pasa como parámetro el cromosoma correspondiente, y se calcula la distancia entre los elementos de ese cromosoma.

Todos los algoritmos, a excepción del Greedy por motivos evidentes, se inicializan con una solución aleatoria.

3. Búsqueda Local

El algoritmo de Búsqueda Local toma una solución inicial y va elaborando vecinos para compararlos con la solución actual y ver cual de las dos soluciones es mejor. Va haciendo esto hasta que realiza 50000 iteraciones o no hay un vecino mejor.
Aquí ha aparecido un nuevo concepto, el de vecino. En este caso un vecino será un conjunto de números idéntico a la solución actual con el siguiente cambio: uno de los elementos de la solución actual será modificado por uno de los elementos que no están en la solución actual.
Para efectuar el algoritmo de Búsqueda Local he seguido los siguientes pasos:

- Calculo el elemento de **mundo** que añade más a la solución actual y lo intercambio de manera aleatoria con uno de los elementos de la solución actual (soy consciente de que debería haber elegido el elemento de la solución que menos aporta e intercambiarlo con un elemento aleatorio de **mundo**, pero considero que este método tiene una mejor solución y el incremento de tiempo no es notable).
    - En caso de que este intercambio no sea beneficioso para la solución actual: Elimino este elemento que pretendía mejorar la solución de **mundo** y lo introduzco en **usados**.

    - En caso de que este intercambio sea beneficioso para la solución actual: Elimino este elemento, que ha mejorado la solución, de **mundo** y limpio **usados** (la operación de intercambio ya elimina de la solución el elemento desechado).
- Esto se repite hasta que se realicen 50000 iteraciones o hasta que **mundo** esté vacío.

Como pseudocódigo se podría ver así:

- Hasta mundo es vacío o se repita 50000 veces:
    - max = elemento de mundo que más aporte a la solución.
    - Si intercambio(max, rand de la solución actual) es beneficioso:
      Elimina rand y añade max.
    - Si intercambio(max, rand de la solución actual) no es beneficioso:
      Añade max a usados.

Para elegir el elemento aleatorio de la solución para intercambiar, simplemente utilizo la función **shuffle** de la biblioteca **algorithm** y tomo el elemento 0.

Ahora voy a complementar la información que he dado sobre la función de intercambio para que se entienda mejor lo que hace. A parte de hacer lo que ya he comentado lo que realiza es una comparación de las puntuaciones de la solución actual y de la solución actual con el intercambio (por medio de una función denominada **remakeScore**).

En caso de que el intercambio no sea beneficioso no hace nada, en cambio, si el intercambio sí es beneficioso para la solución esta misma función se encarga de eliminar el elemento desechado y de introducir el nuevo elemento. Además de cambiar la variable global que controla la puntuación total de la solución actual.

Esta comparación se hace utilizando el método de Factorización del Movimiento de Intercambio, que consiste en no calcular toda la puntuación de la nueva solución, sino en conocer que aporta nuevo la nueva solución y que pierde la solución antigua, y con esto discriminar. De este modo si lo perdido es menor que lo ganado es evidente que la nueva solución es mejor que la antigua y se debe cambiar, en caso contrario, que lo perdido sea mayor que l ganado, la solución antigua debe permanecer.

He elaborado también una función llamada distanciaAcumulada, a la que se le pasa un parámetro, esta obtiene la suma de todas las distancias entre el elemento del parámetro y el resto de la matriz, para así obtener el elemento más distante a todos los elementos de la matriz.

4. Algoritmo Greedy

El algoritmo Greedy se basa en elegir en cada paso el mejor elemento, siendo en este caso el mejor aquel que cumple que esta más alejado de los que están en la solución, aquellos

que están en **mundo**, pues en este algoritmo no se usará **usados**.
Se sigue esta forma de actuar hasta que se llega a tener m elemento en la solución.
Este es un algoritmo sencillo que no tiene mucha explicación, así que esencia eso es lo que he programado:

- Elegir el elemento más distante a la solución actual (en este caso estoy sobrecargando la palabra solución, pues solo podrá considerarse una verdadera solución cuando tenga exactamente m elementos).
- Seguir este proceso hasta que, efectivamente, la solución, tenga exactamente m elementos.

Como pseudocódigo se vería de la siguiente forma:

- Mientras que solucion.tamaño() != m
  - max = elemento de mundo que más aporte a la solución actual.
  - Añade max a la solución actual.

5. Algoritmos Genéticos

Los algoritmos genéticos consisten en tener una población de soluciones, elegir una forma de seleccionar a las que pasarán a poder reproducirse, cruzar aquellas que deban hacerlo, mutar alguna de las soluciones resultantes y repetir este proceso.

En nuestro caso hay dos formas de seleccionar:

- Elitismo: Se trata de seleccionar (mediante un torneo binario) a una cantidad de padres igual a la cantidad de soluciones actuales.
- Esquema Estacionario: Consiste de seleccionar dos padres (con torneo binario).

Una vez seleccionados los padres, los cruzaremos con una probabilidad. En caso de que sea el esquema estacionario la probabilidad debe ser 1. Los cruces que vamos a tener en cuenta son los siguientes:

- Cruce uniforme: En este cruce cada dos padres obtendremos un único hijo.
  Inicialmente, nos quedamos en el hijo aquellas posiciones en las que ambos padres coincidan, y en el resto se le introduce un booleano aleatorio.
  Tras esto cabe se puede dar el caso de que el número de verdaderos (unos) haya variado. De modo que hay que corregir esto, se hace con un reparador.
  En pseudocódigo sería de la siguiente manera:
  - Mientras unos != m
    - Si unos <m
      Añade el elemento que más contribuya a la solución (de los que no están).

- Si unos >m
  Elimina el elemento que más contribuya a la solución (de los que sí están).

- Cruce Basado en Posición: En este cruce obtendremos dos hijos de cada padre.
  Al igual que en el otro cruce, nos quedamos en los hijos aquellas posiciones en las que ambos padres coincidan. Luego elegimos el resto de booleanos que haya en alguno de los dos padres. E introducimos en los huecos de forma aleatoria estos booleanos en los dos hijos.

Tras usar la selección con elitismo y el cruce correspondiente,quizá no se hayan cruzado todos los seleccionados, se mutarán algunos genes con otra probabilidad. Esto consiste en elegir dos posiciones de un mismo cromosoma con valor distinto y cambiarles el valor. Puesto que los valores deben ser distintos, es posible que haya que elegir muchas posiciones hasta que se encuentren dos distintas. Por esto, para las mutaciones lo que he hecho ha sido tomar un número aleatorio hasta m (llamémosle i), y una de las posiciones que se toma será el i-ésimo verdadero. Así, la mutación será más rápida.
En cambio, al usar la selección con esquema estacionario, se cruzarán solo los dos seleccionados, luego, estos hijos, se introducirán en la población original si son mejores que los peores de la población. Tras esto ya se puede proceder a mutar.

De forma que se obtienen 4 algoritmos genéticos, en mi caso, el que ha proporcionado mejores soluciones ha sido con la selección con elitismo y con el cruce Uniforme. A pesar de tardar un poco más, los resultados eran notablemente mejores que los de los demás.

6. Algoritmos Meméticos

Los algoritmos Meméticos son un híbrido entre los genéticos y la búsqueda local. Cada cierto número de generaciones se aplica, sobre cierta cantidad de soluciones actuales un algoritmo de Búsqueda Local para mejorar esa solución.
En nuestro caso hemos aplicado, utilizando el mejor genético obtenido, los siguientes parámetros:

- Cada 10 generaciones toda la población.
- Cada 10 generaciones una décima parte aleatoria de la población.
- Cada 10 generaciones la décima parte mejor de la población.

Lo que he hecho ha sido una función genérica de algoritmos meméticos, es decir, una única función a la que se le pasa por parámetros la cantidad de generaciones que se van a tener en cuenta (**gen**), la proporción que de soluciones que se van a modificar pasadas el número de generaciones impuesto (**prop**) y si se trata de esa proporción de soluciones mejores o no (**mej**).
Viéndolo como pseudocódigo sería así:

- Hasta que score se visite 50000 veces:

- Si han pasado **gen** generaciones:
  - Si mejor: Aplicar BúsquedaLocal(400) (en nuestro caso) sobre las núme-roSoluciones * **prop** mejores soluciones.
  - Si !mejor: Aplicar BúsquedaLocal(400) (en nuestro caso) sobre número-Soluciones * **prop** soluciones aleatorias.

7. Procedimiento para desarrollar la práctica

Para desarrollar la práctica he utilizado código implementado por mí mismo desde el principio, pues todas las estructuras de datos y procedimientos a elaborar eran suficientemente sencillos como para hacerlos yo.
Para ejecutar la práctica basta con tener en una carpeta llamada Tablas las 30 tablas y compilarlo como un programa normal, se debe poner el siguiente comando:
g++ -o ../BIN/nombre_del_ejeutable -Wall -Ofast practicas.cpp
Las opciones de -Wall y -Ofast son para que se compile con máxima opimización.
Para ejecutarlo se puede hacer de la siguiente manera:
./nombre_del_ejeutable n1 n2 donde **n1** es el número de cromosomas que se van a utilizar para hacer los algoritmos genéticos y **n2** es el número de cromosomas que se van a utilizar para hacer los algoritmos meméticos. En caso de no poner **n1** y **n2** serán por defecto 50.

8. Tabla y análisis de resultados

Algoritmo	**Desviación**	**Tiempo(s)**	**Repeticiones**
Greedy	4.99247	0.00765833	-
BL	0.58839	0.0130114	-
AGG-uniforme	0.0018751	1122.53	23.2667
AGG-posición	0.0777266	682.895	12.1
AGE-uniforme	0.0203347	576.309	45.7333
AGE-posición	0.132005	589.802	6.66667
AM-(10, 1)	0.00556232	95.5335	7.06667
AM-(10, 0.1)	0.00748754	352.909	7.13333
AM-(10, 0.1mej)	0.00891129	799.603	7.13333

Antes de comenzar a discutir sobre los resultados voy a comentar el significado de este nuevo parámetro: **Repeticiones**. Lo que quiere decir es la cantidad de veces que se dio el mejor resultado obtenido en la población final de cromosomas. Es por esto que en los algoritmos Greedy y Búsqueda Local carece de sentido. He añadido este parámetro pues también me parece un buen indicador de la bondad de un algoritmo. Es importante notar que para los algoritmos genéticos la población es de 50 cromosomas y para los meméticos es de 10 (como se indica en la práctica).

Para empezar observamos que los menores tiempos van a favor de los primero algoritmos: Greedy y Búsqueda Local, pero fijándonos en las desviaciones vemos que son las más elevadas con diferencia (sobretodo la del Greedy).

Teniendo en cuenta ahora todos los genéticos es fácil ver que, en resultados, AGG-uniforme es el mejor, (siendo un factor de 100 veces mejor que el peor de los genéticos y un factor de 20 veces mejor que el siguiente mejor). Aunque también vemos que es, con gran diferencia, el más costoso en tiempo.
Aquí he de notar que me ha extrañado el elevado tiempo que he obtenido en los últimos 10 casos de ejemplo, pero teniendo en cuenta que eran casos de n = 2000 y m = 200, y los demás casos era más de *juguete* me ha acabado pareciendo coherente. Además, no se me ha ocurrido ninguna manera de factorizar el cálculo de la función objetivo, como hacíamos en la primera práctica.

Siguiendo con el análisis, a pesar del elevado coste temporal del AGG-uniforme, me ha parecido adecuado considerarlo el mejor. Pues, además de su espectacular desviación típica, la cantidad de repeticiones de la mejor solución en la población es de las mejores de los genéticos.

Inicialmente no he querido comparar los algoritmos genéticos con los meméticos, pues con un simple vistazo es completamente evidente que, a pesar de tener un población de cromosomas 5 veces menor, se obtienen unos resultados, tanto por la desviación típica, como por el tiempo empleado, como por las repeticiones de la mejor solución en la población final, claramente superiores a los de los demás. Lo que hace que sea claro que la fusión de algoritmos sea un concepto muy interesante y a tener en cuenta en otras ocasiones.
Queda comentar también que, según los resultados, el mejor de los meméticos es el primero, el que cada 10 generaciones aplica el algoritmo de Búsqueda Local a toda la población durante 400 iteraciones. Con los resultados en la mano, parece claro que no es adecuado fraccionar la población a la que se le aplica el algoritmo BL, y mucho menos que esa fracción sea la mejor, según parece es mucho mś eficiente en tiempo que toda la población cambie por BL, que tener que elegir a los mejores para cambiar. Esto puede deberse al hecho de que haya soluciones que no sean muy buenas, pero sí muy prometedoras. Al aplicar BL a los mejores estamos eliminando la posibilidad de mejorar estas soluciones prometedoras.
Finalmente, voy a comparar los dos mejores algoritmos en base a estos resultados, que

son AGG-uniforme y AM-(10, 1). A pesar de que el primero tiene una varianza unas 4 veces mejor, y tener un buen cociente de repeticiones de la mejor solución (casi la mitad), el algoritmo más adecuado, en general, es el segundo, pues tiene una tasa de repeticiones de la mejor solución mayor, y, más importante que eso, tarda casi 12 veces menos.
Evidentemente, si no nos importara el tiempo empleado y lo que buscáramos fuera la solución más cercana al óptimo, sin ninguna otra restricción deberíamos elegir el primero.

9. Referencias bibliográficas

Para realizar las prácticas me he basado en las explicaciones de las prácticas y la teoría de la asignatura y en C++ reference para abordar el funcionamiento de la clase vector.