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

Práctica 3: Búsquedas por Trayectorias para el Problema de Máxima Diversidad (MDP)

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

8 de junio de 2019

Índice

1 Descripción del problema 3

2 Descripción de las estructuras de datos 3

3 Búsqueda Local 4

4 Algoritmo Greedy 5

5 Enfriamiento Simulado 6

6 Búsqueda Multiarranque Básica 7

7 Algoritmo GRASP 7

8 Búsqueda Local Reiterada (ILS) 8

9 Procedimiento para desarrollar la práctica 8

10 Tabla y análisis de resultados 8

11 Referencias bibliográficas 9

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.

Para generar las soluciones aleatorias iniciales de los distintos algoritmos que lo requieren he seguido el siguiente proceso:

- Inicialmente voy añadiendo números aleatorios entre 1 y n a un unordered_set hasta que tiene m elementos.
- Luego, con un iterador, introduzco los elementos de este unordered_set en el conjunto de la solución.

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. Enfriamiento Simulado

Antes de comenzar a explicar los detalles de la implementación del algoritmo hay que señalar que tenemos un importante número de variables a tener en cuenta:

- Temperatura inicial, que se inicializa a: $\frac{score(solucion\_aleatoria\_inicial)*\mu}{ln(\phi)}$ siendo μ y ϕ dos parámetros inicializados ambos a 0.3.
- Temperatura final, que se inicia a 0.001.
- max_vecinos, con un valor de 10 * n, siendo n el tamaño del problema.
- max_exitos, que hay que iniciar a 0.1 * max_vecinos.
- Enfriamientos, que hay que iniciar a $\frac{50000}{max\_vecinos}$.
- Beta, que tendrá un valor de $\frac{Temperatura\_inicial - Temperatura\_final}{Enfriamientos*Temperatura\_inicial*Temperatura\_final}$

Inicialmente, elaboramos una solución inicial aleatoria, y, puesto que es la única que tenemos, la consideraremos la solución óptima hasta el momento.
Crearemos una variable, que se llamará temperatura, que estará iniciada a Temperatura_inicial.
Iniciaremos entonces un bucle while, cuya condición para mantenerse dentro será que temperatura sea mayor que Temperatura_final. En este while introduciremos otro bucle, un for esta vez. Este bucle interior pasará por todos los números desde 0 hasta temperatura, asegurándose de que si se alcanzan el máximo de éxitos o de vecinos saldrá de este bucle for. En este segundo bucle elegiremos de forma aleatoria un número de la solución y otro de fuera de ella y los intercambiaremos. Calcularemos la diferencia de costes entre la solución que tengamos guardada como óptima y la vecina mediante la factorización del coste:

- En caso de que esta diferencia sea positiva (la solución nueva es mejor que la óptima) se aumentará el contador de éxitos y se actualizará la solución óptima.
- En caso de que la distribución uniforme entre 0 y 1 sea menor que: $exp(\frac{-diferencia}{temperatura})$, también se ejecutará el caso anterior.
  Esto es lo que diferencia al Enfriamiento Simulado de otros algoritmos, pues en este caso concreto se estaría aceptando una solución peor que la actual, pero esto nos puede ayudar a salir de máximos locales.
- Se acepte la solución vecina o no, habrá que incrementar el contador de vecinos.

Una vez acabado el bucle interior se actualizará la temperatura según el esquema de Cauchy modificado: $temperatura = \frac{temperatura}{1+Beta*temperatura}$, siendo Beta, la variable que he definido con este nombre con anterioridad.
Una vez llegado a este punto, si no se había producido ningún éxito saldremos del bucle exterior, sino seguiremos disminuyendo la temperatura.

6. Búsqueda Multiarranque Básica

Este es un algoritmo muy sencillo, únicamente consiste en elegir 25 soluciones iniciales (en este caso), elegidas de forma aleatoria, y aplicarles a todas ellas el algoritmo de Búsqueda Local, nos quedaremos con el mejor resultado de todos ellos.

7. Algoritmo GRASP

El algoritmo GRASP es una variante del conocido algoritmo Greedy.
Mientras que en el algoritmo Greedy elegimos el elemento más a alejado de todos para empezar, y luego continuamos añadiendo elementos que estén alejados de nuestro proyecto de solución, hasta que este tiene m elementos. El algoritmo GRASP tiene el mismo pimer paso, pero luego, en lugar de añadir el elemento más alejado hace una lista de candidatos a ser una buena solución y añade uno aleatorio de ellos. Permitiendo así que se empeore al principio, pero pudiendo mejorar después.
Para controlar esta lista de candidatos he decidido tomar un vector de parejas, siendo el primero de la pareja la distancia a los elementos de la solución y el segundo la posición de que tiene. Así, cuando tenga que mutilar la lista podré recuperar tanto la distancia de los elementos como la posición que tenían. Esta mutilación de la lista se hace de la siguiente manera: debemos generar, en cada iteración, una distancia mínima, que, en este caso, será:
Distancia_mínima = distancia_menor + alfa * (distancia_mayor - distancia_menor), siendo este alfa un parámetro, que en este caso se nos ha dicho que sea 0.3. Como dato de interés, este proceso también se puede hacer con una distancia mínima fija, pero lo que hemos hecho nosotros parece mejor, pues con ello se ajusta la distancia al momento del problema, y no al problema en sí. Si tuviéramos que dar una distancia fija deberíamos cambiarla para cada caso.

Como pseudocódigo se haría de la siguiente forma:

- Añadimos a nuestro proyecto de solución el elemento más alejado de todos.
- Mientras el tamaño de la solución es menor que m:
  - Elaboro una lista con todos los elementos de fuera de la solución con sus distancias respectivas.
  - Aplico el criterio de la distancia mínima y elimino los elementos que no la pasan.
  - Añado al proyecto de solución uno de los elementos de esta lista de forma aleatoria.
- Una vez que el proyecto de solución ya se ha convertido en una solución real, es decir, ya tiene m elementos, le aplicamos el algoritmo de Búsqueda Local.

Este algoritmo lo aplicamos 25 veces, en nuestro caso, y nos quedamos con el mejor resultado.

8. Búsqueda Local Reiterada (ILS)

Este algoritmo consiste en elaborar una solución inicial aleatoria, aplicarle Búsqueda Local y guardar esta solución como la óptima hasta el momento. Luego, repetir el siguiente proceso la cantidad de veces que se desee, en nuestro caso serán 25 veces:

- Aplicar una función de mutación que cambie radicalmente la solución, de la siguiente forma:
  - En lugar de elegir un nodo y reasignar los elementos de desde este hasta el tamaño deseado como se propone en el guión he preferido hacer lo siguiente. He elegido una cantidad de elementos del interior de la solución (que en nuestro caso será de una décima parte del tamaño de la misma) y los he intercambiado con esa misma cantidad de elementos de fuera de la solución.
    He decidido tomar esta decisión, pues el incremento en tiempo no era exageradamente elevado y porque considero que así existe un mayor grado de mutación en nuestra solución.
- Una vez realizada la mutación, volvemos a utilizar el algoritmo de Búsqueda Local y comparamos con el óptimo actual.

Finalmente, se devolverá última solución óptima.

9. 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.

10. Tabla y análisis de resultados

Algoritmo	**Desviación**	**Tiempo(s)**
Greedy	4.99247	0.00765833
BL	0.58839	0.0130114
ES	0.154164	0.0533024
BMB	0.551906	0.0310311
GRASP	0.255832	11.3529
ILS	0.159357	7.3547
ILS-ES	0.133589	0.68593

Como podemos ver, el algoritmo Greedy sigue siendo el que gasta menor cantidad de tiempo, pero teniendo una desviación unas 10 veces mayor que el siguiente algoritmo con mayor desviación.

Los dos siguientes algoritmos por orden de desviación son la Búsqueda Local y la Búsqueda Multiarranque Básica. Es lógico que la desviación de BMB sea menor que la de BL, pues, en esencia, son el mismo algoritmo, tan solo que BMB hace lo mismo que BL, pero 25 veces y quedándonos con la mejor solución. Es por esto, que, de forma evidente, el coste en tiempo de BMB es mayor que el de BL.

Siguiendo con este orden, el algoritmo a analizar ahora será el algoritmo GRASP. Este, tiene una desviación que se queda en la media de los algoritmos de esta práctica, pero tiene el coste en tiempo más elevado de todos ellos. Esto se debe a que tiene que elaborar una cantidad muy elevada de listas de candidatos. Aunque en los casos pequeños no es de gran importancia, en los casos grandes hay que fabricar muchas listas de tamaños muy elevados.

Tenemos también los algoritmos Enfriamiento Simulado y Búsqueda Local Reiterada, que tienen unos resultados realmente buenos en comparación con los demás. Con respecto al tiempo que tarda cada uno vemos que ES tarda mucho menos que ILS, esto se debe a que ES profundiza con mucha rapidez en general. El tiempo de ILS es tan elevado por la decisión que he comentado a la hora de explicar ILS. A pesar del tiempo que tarda, no considero que sea excesivo, y nos proporciona unos resultados bastante buenos.

Finalmente, al hibridar los algoritmos ES e ILS (los que tienen mejores resultados), obtenemos uns resultados tímidamente mejores ambos por separado, aunque en tiempo, tarda un poco más que ES. Esta hibridación consiste en aplicar el algoritmo ILS, pero, cuando invocábamos la Búsqueda Local para mejorar nuestra solución, llamamos a ES para mejorarla aún más.

En definitiva, después de comentar los resultados de los algoritmos divididos por clases, vemos que los mejores resultados se apoyan en el Enfriamiento Simulado, tanto en buenos resultados, como en buen tiempo. Características esenciales para determinar un buen algoritmo sobre unos datos.

11. 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 y los pair.