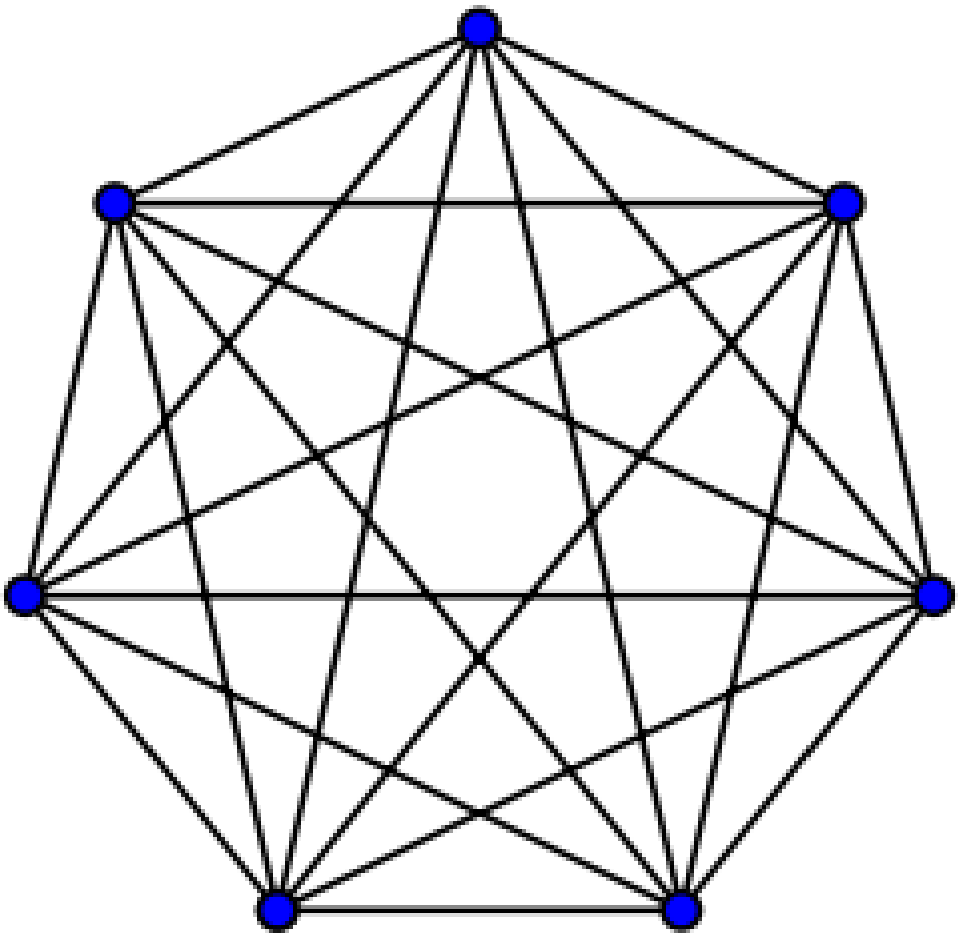
Diseño y Análisis de Algoritmos

Max-Mean Dispersion Problem



Esther Jorge Paramio

Ingeniería informática. 3º Curso.

Itinerario de computación.

Introducción

El objetivo de esta práctica es diseñar e implementar diversos algoritmos constructivos y búsquedas por entorno. En este caso, hemos diseñado los diferentes algoritmos para un problema de maximización, en concreto el Max-Mean Dispersion Problem, que dado un grafo completo, se desea encontrar el subconjuntos de vértices que maximiza la dispersión media dada por:

$$md(S) = \sum d(i, j) / |S|$$

Las instancias del problema se suministrarán en un fichero de texto con el siguiente formato: en la primera fila se encuentra el número de vértices, a continuación, se enumeran las afinidades, d(i, j), entre los pares de vértices (se asume que las afinidades son simétricas, es decir, que d(i, j) = d(j, i), $\forall$ i, j $\in$ V. Además, d(i, i) = 0, $\forall$ i $\in$ V).

Arquitectura

- Procesador: Intel(R) Core(™) i7-8750H CPU @ 2.20GHz 2.2 GHz
- RAM: 8,00 GB
- Tipo de sistema: 64 bits
- Sistema Operativo: Windows 10s

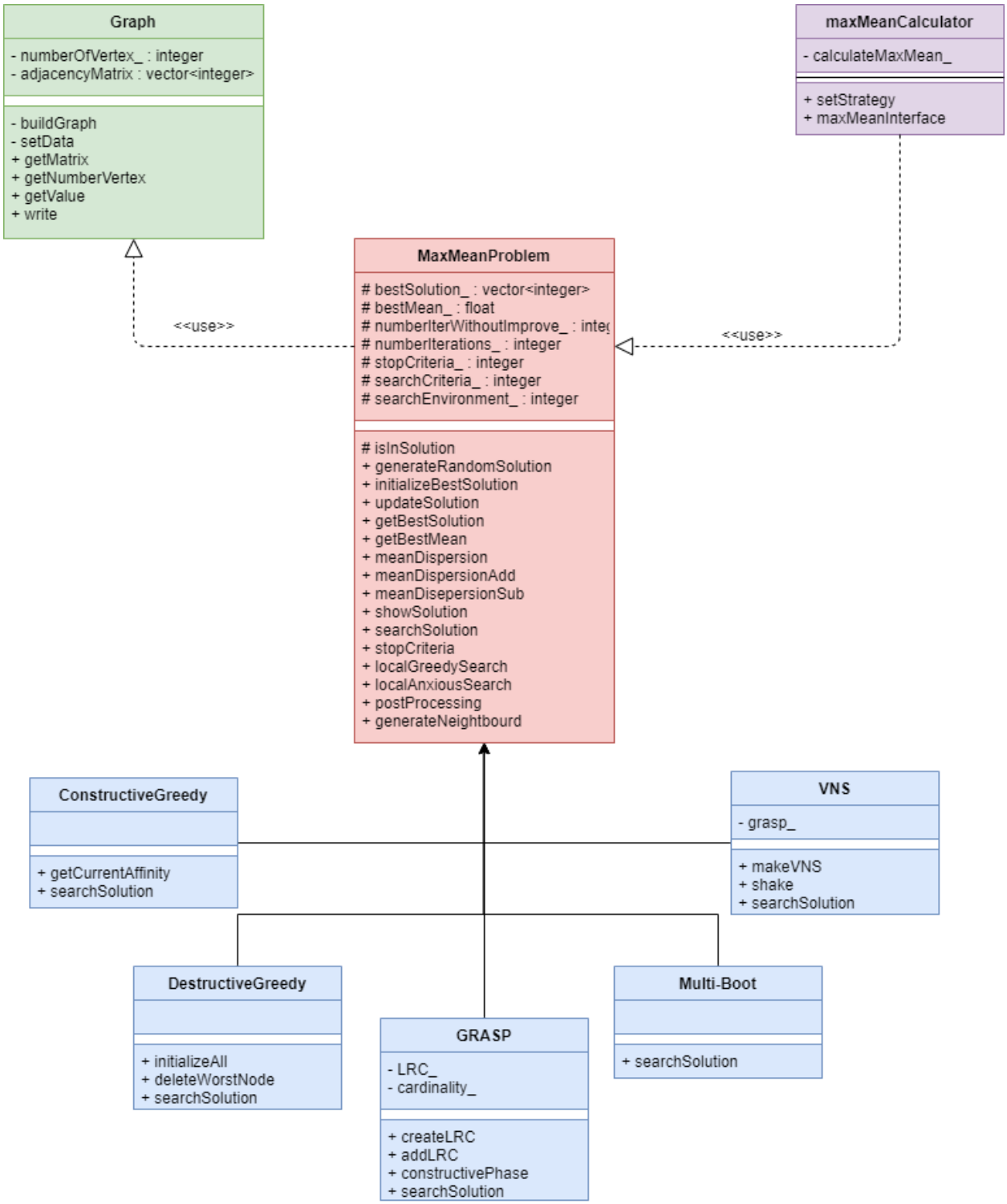
Estructura del código

Para implementar los diferentes algoritmos, he decidido utilizar el patrón estrategia para poder agilizar la implementación de todos estos en una sola ejecución en un mismo programa.

La estructura es la siguiente:

- Include: Esta carpeta contiene todas las cabeceras que se ha utilizado para desarrollar el código.
    - MaxMeanStrategyBase.hpp: clase padre de la estrategia de la que heredarán los algoritmos a implementar para poder utilizar una interfaz común.
    - Graph.hpp: clase que se encarga de manejar la matriz adyaciencia del grafo dado por un fichero.
    - El resto de los *.hpp*, son las cabeceras heredadas de de la clase base del problema *MaxMean* que implementarán los diferentes algoritmos de la búsqueda de soluciones.
- Src: Esta carpeta contiene todos los ficheros *.cpp* con el código de las clases además del programa principal.
- Bin: Contiene el fichero ejecutable.
- En la raíz de encuentran los ficheros con los grafos y el *Makefile.*

Diagrama UML



Explicación de los algoritmos

Greedy Constructivo

Para el diseño de este algoritmo he utilizado el pseudo-código dado en el guion. Al ser un algoritmo constructivo partimos desde una solución vacía.

El método que va a ser llamado para buscar la mejor solución, *seachSolution*, antes que nada va inicializar la solución con los nodos con la arista con mayor afinidad. Esto es porque como mencioné antes se va a partir del conjunto vacío, por eso se necesita este paso previo antes de iterar para buscar la mejor solución, ya que al ser un algoritmo voraz se va a elegir la solución que maximice el problema en ese momento independientemente de las decisiones anteriores o siguientes sin volver nunca atrás. En este caso, serán los nodos inicializados con la arista con mayor media y en caso de que haya dos o más nodos con la misma media se escogerá uno de ellos de forma aleatoria.

A continuación, hasta que la solución no mejore en la siguiente iteración, se van añadiendo iterativamente los vértices que producen el mayor incremento positivo a la función objetivo, en este caso, mayor media. Para ello, en vez de recalcular a fuerza bruta la media desde 0 con la nueva solución, llamamos a una función que nos añade a la media calculada previamente el peso del nuevo nodo añadido. Como estamos calculando media, lo que hacemos es calcular la suma del coste de las aristas del nodo que queremos calcular con los nodos que ya se encuentran en la solución, luego simplemente multiplicamos la media calculada con el número totales de nodos en la solución más el coste del nodo que queremos añadir y todo esto lo dividimos entre el tamaño de la solución contando con el nuevo nodo.

Por último, comprobamos si la solución calculada es mejor que la anterior, si esto se cumple actualizamos y seguimos buscamos, de lo contrario para de iterar y devuelve esa solución.

Greedy Destructivo

Este segundo algoritmo, al ser greedy, sigue el mismo proceso que el mencionado anteriormente, salvo que en este caso en vez de ser constructivo es destructivo. Esto quiere decir que a diferencia del otro, en vez de partir con la solución vacía e ir añadiendo los nodos que maximicen la función objetivo, inicializamos la solución con todos los nodos y vamos eliminando aquel que tenga la arista con menor afinidad.

Para implementarlo, primero que nada, nuestra función *searchSolution*, inicializará todos los nodos y calculará la media a partir de esa solución. Luego, mientras se actualice la solución respecto la anterior, seguimos eliminando los nodos que empeoran nuestra función objetivo.

GRASP

Este algoritmo también es constructivo y utiliza una estrategia voraz, la diferencia es que va a introducir aleatoriedad a la búsqueda de la solución. Para ello vamos a tener una lista restringida de candidatos, la cual va a contener los mejores nodos del cual se escogerá uno al azar y se añadirá a la solución. Para ello, el GRASP se divide en varias fases.

Primero, se encuentra la fase de preprocesamiento, al ser constructivo, inicializamos la solución igual que en el greedy constructivo, con los nodos con la mayor arista.

Luego, viene la fase constructiva, que se encargará de generar la mejor solución a partir del LRC. Esta lista, según lo visto en clase, la podemos crear de tres formas diferentes, por cardinalidad, rango o intersección entre estas dos anteriores. En mi caso, he optado por la primera, en la cual fijo una cardinalidad, que será el número de nodos que contendrá nuestro conjunto y añado aquellos que maximicen la media. Volviendo a la fase constructiva, lo que haremos será crear una solución parcial al igual que en el algoritmo constructivo voraz, salvo que en vez de añadir el nodo que maximice la media según la solución actual, elegimos un nodo al azar de los que se encuentra en nuestro conjunto LRC.

Una vez generada la solución parcial en la fase constructiva, realizamos la siguiente fase, el postprocesamiento, cuyo objetivo será mejorar las soluciones obtenidas en la anterior fase. Para ello, hemos implementado dos búsquedas locales diferentes que explicaré más adelante en este documento, una greedy y otra ansiosa. Además, estas búsquedas pueden usar dos estructuras de entorno diferentes, de apertura y de cierre, según como se indique.

Por último, si la nueva solución creada en la fase constructiva es mejor actualizamos, si no, seguimos iterando hasta que se cumpla el criterio de parada, el cual también se han implementado dos y según el que se especifique se usa uno u otro. Estos criterios se cumplen cuando se alcance un número máximo de iteraciones establecidas por el usuario o un número máximo de iteraciones sin mejora.

Multi-Arranque

Este algoritmo desarrolla búsquedas locales desde varias soluciones de inicio, preferiblemente uniformemente distribuidas por el espacio de soluciones, hasta que se cumpla un criterio de parada. Para ello, primero determinaremos qué mecanismo de generación de soluciones utilizar, hay varias formas, podemos utilizar GRASP, pero en ese caso la búsqueda local y/o la estructura de entorno deberían ser diferentes al implementado en este último, ya que si no darían siempre la misma solución, en este caso he optado por generar las soluciones iniciales de forma totalmente aleatoria pero implementando la misma estructura de entorno y búsqueda local que he implementado en el algoritmo de GRASP.

Una vez generada nuestra solución inicial de la forma que hayamos elegido, realizamos el post-procesamiento, que sería realizar la búsqueda local al igual que GRASP. Comprobamos si esta nueva solución es mejor a la actual, y si es así actualizamos y si no sumamos uno al contador de generación de soluciones sin mejoras y volvemos a generar otra solución hasta que se cumpla el criterio de parada establecido.

VNS

Este algoritmo es bastante simple, pero aún así es uno de los que más se suele utilizar por sus buenos resultados en variados tipos de problema.

Este algoritmo se basa en el principio que comentamos antes, las mejores soluciones suelen estar cerca unas de otras. Básicamente, lo que se hace es buscar en diferentes estructuras de entorno el óptimo local de todas esas. Para ello primero se selecciona una estructura y se busca el óptimo local de esta, cuando lo tenga se estudia la siguiente estructura, que deberá contener a la anterior, y si el óptimo local cambia, entonces se vuelve a realizar la búsqueda con la nueva solución iterando desde la primera estructura. Esto se repite hasta que buscando en los diferentes entornos el óptimo local no cambie, llegando a la conclusión que es el mejor contenido en todas ellas.

En la práctica, primero creamos una solución con GRASP, aunque para que se note mejor el cambio podemos generar soluciones aleatorias, ya que GRASP de por sí suele dar resultados bastante buenos. Luego, y mientras no se cumpla el criterio de parada, llamamos a una función auxiliar que nos va a hacer el VNS, en ella vamos a recorrer distintas estructuras de entorno. En este caso, he optado por buscar en tres estructuras de entorno diferentes con movimiento de intercambio.

Por consiguiente, se llama al método *shake*, que calculará una vecina del entorno de forma aleatoria. Esta vecina estará contenida en el anillo que se forma entre el entorno actual y los entornos ya estudiados. Por último, se realiza el post-procesamiento de la solución aleatoria generada. En el caso de encontrar una solución mejor, buscamos desde el primer entorno con esa solución, si no, avanzamos al siguiente entorno hasta que ya se hayan recorrido todos y no se encuentre una solución mejor.

Este proceso se repite hasta que se cumpla el criterio de parada.

Búsquedas locales

Las búsquedas locales las realizamos porque hay un estudio empírico que argumenta que las buenas soluciones suelen estar cercanas en el espacio. Para ello, lo que realizamos en esta búsqueda es generar las soluciones vecinas a la solución actual para comprobar si se encuentra una vecina mejor que la actual.

Para ello, primero generamos una solución, que puede ser determinista por ejemplo el greedy, aleatorio como el multi-arranque o mixtos, que combinan los dos anteriores, como el GRASP.

Luego, generamos la solución vecina dependiendo de la estructura de entorno que hemos fijado. Los movimientos que hemos dado en clase han sido tres, por apertura, cierre o intercambio. En el código he implementado dos movimientos, el de apertura al que le añadimos un nodo que no pertenezca a la solución y de cierre, al que eliminamos un nodo perteneciente a la solución actual.

Por último, comprobamos si la solución vecina es mejor que la actual y actualizamos si esto se cumple. Repetimos esta acción hasta que se cumplea el criterio de parada.

En muchas ocasiones, se intenta una búsqueda del entorno actual en busca de la mejor vecina siendo esto muy ineficiente en muchas ocasiones. Por ello, se consideran otros muestreos del entorno. En este caso, hemos implementado dos, la búsqueda greedy que realiza una búsqueda exhaustiva por el entorno generando todas las vecinas y devolviendo la mejor y la búsqueda ansiosa, que finaliza en cuanto se encuentra una solución mejor que la actual.

Tablas

- **Greedy constructivo**

Problema	Nº vértices	Ejecución	Mejor Media	CPU ms
div-10	10	1	11.8571	6.8975
div-10	10	2	10.1429	6.9955
div-10	10	3	13	6.5066
div-10	10	4	14	4.3152
div-10	10	5	13	4.2166
div-15	15	1	9.5	4.7227
div-15	15	2	7.5	3.7613
div-15	15	3	7.5	4.8894
div-15	15	4	9.75	8.7152
div-15	15	5	9.75	2.5473
div-20	20	1	13.1667	7.1448
div-20	20	2	13.1667	3.6915
div-20	20	3	9.75	3.4669
div-20	20	4	13.1667	7.1177
div-20	20	5	8.8	3.3468

- **Greedy destructivo**

Problema	Nº vértices	Ejecución	Mejor Media	CPU en ms
div-10	10	1	7.9	7.1048
div-10	10	2	7.9	6.0618
div-10	10	3	9	5.4783
div-10	10	4	7.9	5.4262
div-10	10	5	9	4.5977
div-15	15	1	-3.76923	7.4229
div-15	15	2	-8.26667	3.4448
div-15	15	3	1.72917	5.528
div-15	15	4	-0.416667	3.9848
div-15	15	5	-8.26667	3.8687
div-20	20	1	-3.9	1.0505
div-20	20	2	-3.9	3.5309
div-20	20	3	-3	4.4371
div-20	20	4	-3.57895	5.3146
div-20	20	5	-1.41176	5.4382

- **GRASP**

Problema	Nº vértices	Max iter	Local Search	|LRC|	Ejecución	Mejor Media	CPU
div-10	10	50	Anxious	4	1	12.8	0.6105
div-10	10	50	Anxious	4	2	13	6.5706
div-10	10	50	Greedy	4	3	14	6.2229
div-10	10	50	Anxious	2	1	14	2.5896
div-10	10	50	Anxious	2	2	14	10.2588
div-10	10	50	Greedy	2	3	14	2.6198
div-15	15	50	Anxious	4	1	9.33333	4.0115
div-15	15	50	Anxious	4	2	9.75	3.4234
div-15	15	50	Greedy	4	3	9.83333	4.7535
div-15	15	50	Anxious	2	1	9.75	6.9244
div-15	15	50	Anxious	2	2	9.83333	5.4182
div-15	15	50	Greedy	2	3	9.75	5.8673
div-20	20	50	Anxious	4	1	13.1667	4.1711
div-20	20	50	Anxious	4	2	13.1667	5.8135
div-20	20	50	Greedy	4	3	13.1667	7.2533
div-20	20	50	Anxious	2	1	13.1667	8.2377
div-20	20	50	Anxious	2	2	13.1667	9.4332
div-20	20	50	Greedy	2	3	13.1667	9.7772

- **Multi-Arranque**

Problema	Nº vértices	Max iter	Local Search	Ejecución	Mejor Media	CPU
div-10	10	50	Anxious	1	10.7143	22.8868
div-10	10	50	Anxious	2	11.6	17.4048
div-10	10	50	Anxious	3	12.8	20.4386
div-10	10	50	Greedy	1	14	13.7728
div-10	10	50	Greedy	2	14	19.8002
div-10	10	50	Greedy	3	14	28.9191
div-15	15	50	Anxious	1	6.16667	21.6157
div-15	15	50	Anxious	2	8.75	22.0394
div-15	15	50	Anxious	3	6.25	19.2557
div-15	15	50	Greedy	1	7.4	44.402
div-15	15	50	Greedy	2	9.75	36.8338
div-15	15	50	Greedy	3	7.5	35.2017
div-20	20	50	Anxious	1	4.25	17.1649
div-20	20	50	Anxious	2	7	18.4512
div-20	20	50	Anxious	3	7.2	29.7539
div-20	20	50	Greedy	1	13.1667	59.514
div-20	20	50	Greedy	2	12.5714	48.6937
div-20	20	50	Greedy	3	10.8	50.1203

- **VNS**

Problema	Nº vértices	k_{max}	Ejecución	Mejor Media	CPU en ms
div-10	10	3	1	14	36.1396
div-10	10	3	2	14	28.5756
div-10	10	3	3	14	36.769
div-10	10	3	4	14	24.3805
div-10	10	3	5	24.3805	38.8007
div-15	15	3	1	9.75	26.8677
div-15	15	3	2	9.75	27.9886
div-15	15	3	3	9.83333	32.9853
div-15	15	3	4	9.83333	35.8396
div-15	15	3	5	9.83333	38.9143
div-20	20	3	1	13.1667	32.1815
div-20	20	3	2	13.1667	49.9103
div-20	20	3	3	13.1667	13.1667
div-20	20	3	4	13.1667	44.2648
div-20	20	3	5	13.1667	32.6362