Max mean dispersion problem.

Diseño y análisis de algoritmos.

Alumno: Sabato Ceruso.

Índice de contenido

Introducción

Max-Mean dispersion problem.

Se pretende realizar una implementación de diversos algoritmos para la resolución del problema del problema de las k medias y del posterior estudio del comportamiento de estos algoritmos.

Concretamente se hizo una implementación de dos algoritmos GRASP constructivos distintos, uno que partiendo de una solución muy pequeña añade nodos uno a uno, y otro que partiendo de todos los nodos elimina nodos uno a uno. También se implementó dos algoritmos Greedy, siguiendo la misma filosofía que antes; un GRASP con post-procesamiento, un multiarranque y una búsqueda por entornos variable.

A continuación se explicara cada uno de estos algoritmos, seguido de una explicación de la estructura del programa y estructuras de datos empleadas y finalmente se detallará la experiencia computacional realizada.

Algoritmos

Algoritmos Greedy.

Se implementó, como se ha mencionado antes, dos algoritmos greedy, uno que como solución inicial parte del par de nodos con mayor afinidad, y, va agregando nodos hasta que no encuentra ningún nodo que consiga dar una mejor solución que la que tiene; y, otro que parte desde una solución con todos los nodos y elimina hasta que se encuentra con el mismo criterio de parada.

En la práctica, estos algoritmos se implementaron como casos especiales de la fase constructiva del algoritmo GRASP pero con la particularidad de poseer una lista restringida de candidatos de tamaño 1, de este modo, se asegura que en cada iteración se escoge siempre el mejor candidato.

Algoritmos GRASP.

Los algoritmos GRASP se diseñaron permitiendo cambiar el tamaño de la lista restringida de candidatos a conveniencia. El tamaño de esta lista deberá ser fijado antes de cada ejecución y será un número fijo, el tamaño real de la lista podrá ser menor pero nunca mayor.

A la hora de seleccionar una solución de la lista de candidatos se selecciona realizando una tirada completamente aleatoria entre las soluciones mejores.

En la fase de post-procesamiento, se realiza una búsqueda local utilizando como entorno aquellas soluciones que pueden ser producidas como resultado de insertar o eliminar un único elemento. El muestreo del entorno se realizará, en este algoritmo y en los demás, de forma ansiosa, con el fin de evitar recorrer todo el entorno de la solución y además, no se podrá pasar a una solución de peor calidad de la que ya se tiene. El criterio de parada será el encontrar un óptimo local.

Método multiarranque.

Para la implementación del método multiarranque se generó las soluciones de forma completamente

aleatoria para luego realizar una búsqueda local igual a la realizada anteriormente en el post-procesamiento de los algoritmos GRASP. El criterio de parada utilizado fue el alcanzar un determinado número de iteraciones sin mejora.

Se ha hecho el código de tal forma de que sea muy sencillo utilizar un sistema distinto de generación de soluciones, como puede ser mismo un GRASP, así como cambiar el criterio de parada o la búsqueda local que se utiliza.

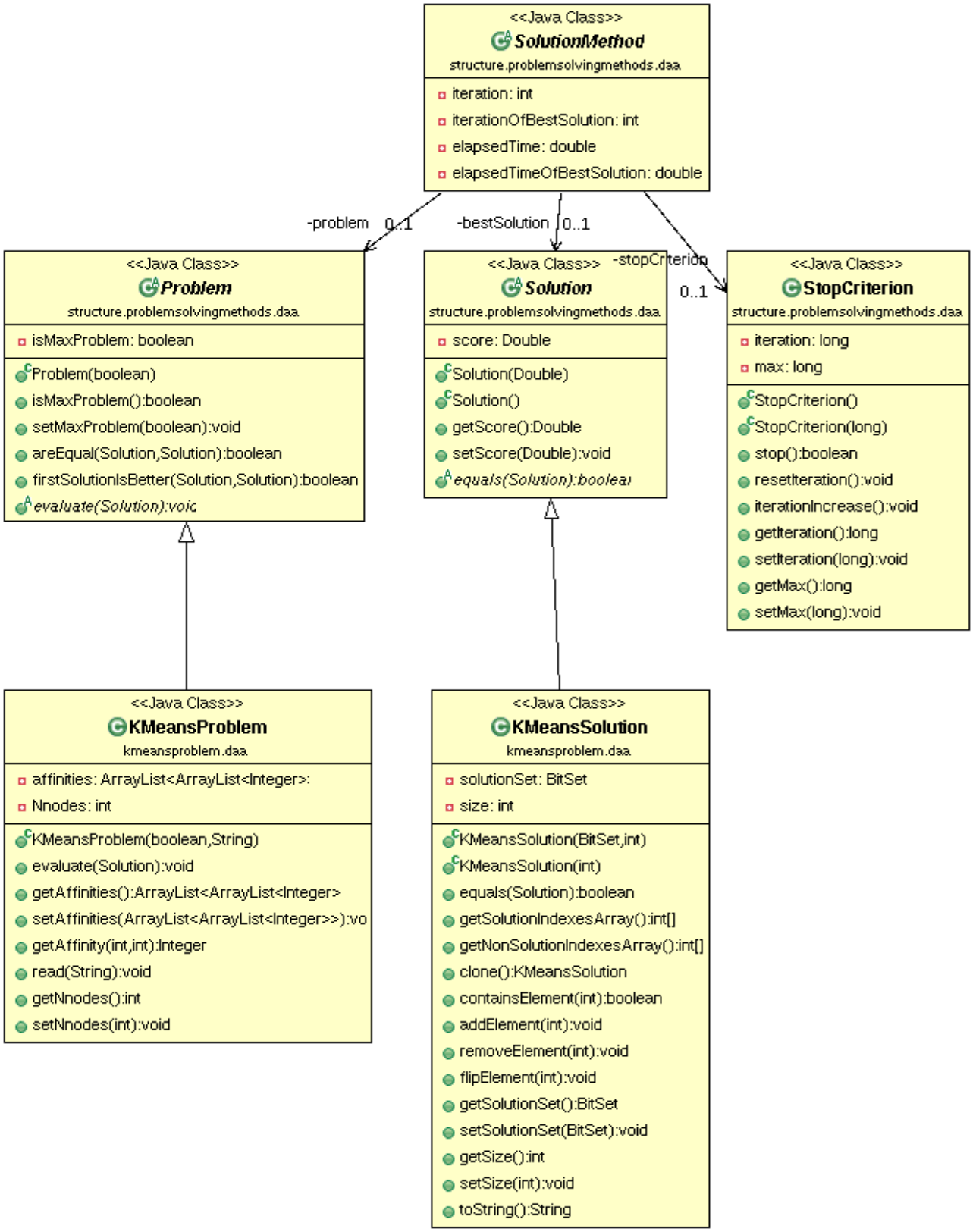
Búsqueda por entorno variable.

Para el VNS se utilizó, en primer lugar una generación aleatoria de la solución inicial. Para la lista de entornos se estableció un mínimo de 3 entornos que son aquellos formados por las soluciones resultantes de realizar los movimientos de eliminar un elemento, insertar un elemento e intercambiar un elemento respectivamente. También se puede ejecutar el algoritmo con más entornos, que serán aquellos formado por el conjunto de soluciones que son resultado de intercambiar 2 elementos de la solución, intercambiar 3, intercambiar 4... y así sucesivamente. La búsqueda local que se realiza sobre las soluciones es una búsqueda local sobre el entorno al que pertenece la solución encontrada.

Estructuras utilizadas.

Para la implementación se realizó una estructura de clases donde hay una clase abstracta problema que representa todos los problemas, una clase abstracta solución que representa las soluciones a un problema abstracto y una clase abstracta método de solución que representará los algoritmos a utilizar para resolver un determinado problema.

A continuación se extiende las clases de problema y solución para dar forma a nuestro problema en particular. A continuación se muestra el diagrama de clases que se acaba de explicar.


<<Java Class>>
SolutionMethod
structure.problemsolvingmethods.daa
iteration: int
iterationOfBestSolution: int
elapsedTime: double
elapsedTimeOfBestSolution: double
-problem
0..1
-bestSolution
0..1
-stopCriterion
0..1
<<Java Class>>
Problem
structure.problemsolvingmethods.daa
isMaxProblem: boolean
Problem(boolean)
isMaxProblem():boolean
setMaxProblem(boolean):void
areEqual(Solution,Solution):boolean
firstSolutionIsBetter(Solution,Solution):boolean
evaluate(Solution):voic
<<Java Class>>
Solution
structure.problemsolvingmethods.daa
score: Double
Solution(Double)
Solution()
getScore():Double
setScore(Double):void
equals(Solution):boolea
<<Java Class>>
StopCriterion
structure.problemsolvingmethods.daa
iteration: long
max: long
StopCriterion()
StopCriterion(long)
stop():boolean
resetIteration():void
iterationIncrease():void
getIteration():long
setIteration(long):void
getMax():long
setMax(long):void
<<Java Class>>
KMeansProblem
kmeansproblem.daa
affinities: ArrayList<ArrayList<Integer>:
Nnodes: int
KMeansProblem(boolean,String)
evaluate(Solution):void
getAffinities():ArrayList<ArrayList<Integer>
setAffinities(ArrayList<ArrayList<Integer>>):vo
getAffinity(int,int):Integer
read(String):void
getNnodes():int
setNnodes(int):void
<<Java Class>>
KMeansSolution
kmeansproblem.daa
solutionSet: BitSet
size: int
KMeansSolution(BitSet,int)
KMeansSolution(int)
equals(Solution):boolean
getSolutionIndexesArray():int[]
getNonSolutionIndexesArray():int[]
clone():KMeansSolution
containsElement(int):boolean
addElement(int):void
removeElement(int):void
flipElement(int):void
getSolutionSet():BitSet
setSolutionSet(BitSet):void
getSize():int
setSize(int):void
toString():String

Para la implementación de los algoritmos, cada uno de ellos extenderá la clase SolutionMethod, pues, cada uno de ellos será un método para proporcionar una solución.

En primer lugar, la clase ConstructiveGRASP se ocupará de generar una solución siguiendo un método GRASP, a ella la extienden DirectConstructiveGRASP y ReverseConstructiveGRASP que se ocupan de generar una solución añadiendo o quitando nodos siguiendo un método GRASP.

El algoritmo Greedy estará compuesto por un ConstructiveGRASP con un tamaño de lista restringida de candidatos igual a 1 y simplemente se ocupará de crear dicho GRASP y ejecutarlo.

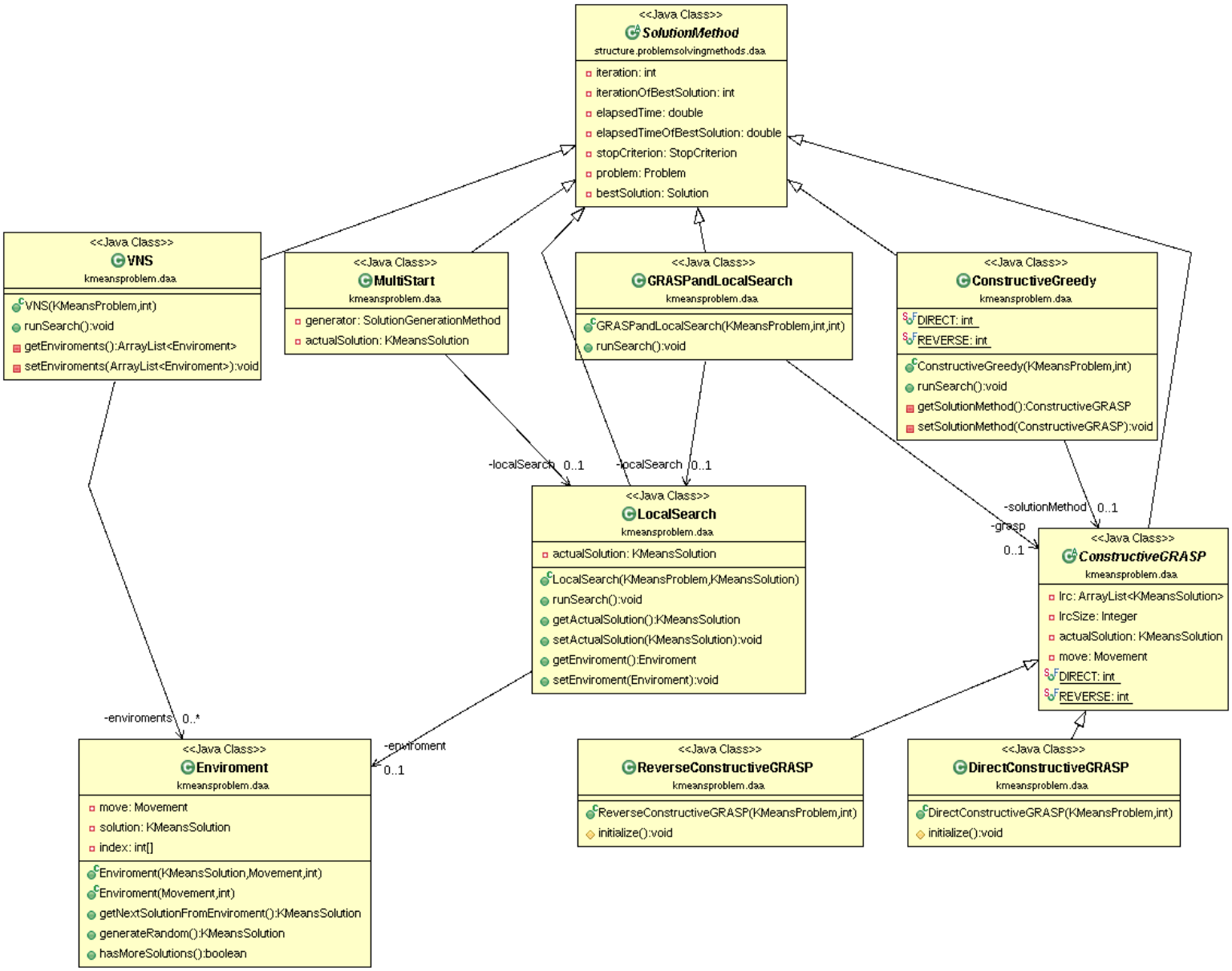
Luego se implementó la búsqueda local, donde se realizará una búsqueda a partir de una solución utilizando una estructura de entorno dada. Para realizar esta búsqueda se definió primero la clase Enviroment, que define a un entorno, según esta estructura, un entorno está definido por un movimiento (agregar, quitar o intercambiar) y por un índice que indica el número de nodos a los que se le puede aplicar dicho movimiento, o, en otras palabras, la diferencia máxima que puede haber entre la solución y su entorno. Por ejemplo, un entorno de intercambiar de índice 2, para la , solución {1, 3, 7}, las soluciones {1, 2, 3, 4, 7}, {3, 5, 7}, {1} estarían en su entorno pero no la {5, 7}.

Una vez que se tiene la búsqueda local, se pudo implementar la clase GRASPandLocalSearch que se ocupa de realizar una construcción mediante un método GRASP de una solución para luego realizar una búsqueda local sobre dicha solución. Esta clase es muy simple, está compuesta por un ConstructiveGRASP y un LocalSearch y solamente se ocupa de ejecutar ambos algoritmos uno tras otro.

El algoritmo multiarranque se implementó con la clase Multistart, está tiene un generador de soluciones aleatorio, un LocalSearch y un StopCriterion, el funcionamiento es simple, genera una solución aleatoria y la mejora con la búsqueda local mientras el criterio de parada no pare el algoritmo.

Finalmente, se implementó el VNS, este algoritmo utiliza una lista de entornos de diferentes dimensiones y movimientos, y una búsqueda local, para su funcionamiento, primero genera de forma aleatoria una solución del primer entorno y la mejora, si no consigue que sea mejor que la que ya tiene como mejor, entonces generará otra solución de forma aleatoria pero del entorno siguiente. Si llega al ultimo entorno y no consigue mejorar la mejor entonces para.

A continuación se muestra el diagrama UML de clases de los algoritmos:



Experiencia computacional

Sistema sobre el que se realiza:

SO: Ubuntu.

Procesador: Intel core i5.

RAM: 6 GB.

Algoritmo Greedy inverso:

Tamaño (n)	Valor objetivo	Iteraciones	Tiempo (Milis)
10	14	5	4
15	9,83	10	7
20	12,1428	14	11
50	46,73	21	53
100	78,34	19	125

Algoritmo Greedy directo:

Tamaño (n)	Valor objetivo	Iteraciones	Tiempo (Milis)
10	10,14	6	2
15	9,5	3	2
20	12,85	6	4
50	25	1	2
100	50	1	4

Algoritmo GRASP inverso con búsqueda local:

Tamaño (n)	LRC	Valor objetivo	Iteraciones	Tiempo (Milis)
10	2	14	14	4
10	3	14	14	5
10	4	14	14	6
20	2	12,85	33	12
20	3	12	33	12
20	4	12,85	58	12
50	2	46,73	70	43
50	3	46,73	70	48
50	4	46,73	70	65
100	2	78,38	189	144
100	3	77,709	114	118
100	4	78,55	400	133

Algoritmo multiarranque.

Tamaño (n)	Valor objetivo	Iteraciones	Tiempo (Milis)
10	14	24	17
15	9,83	21	18
20	13,16	21	22
50	46,73	23	168
100	78,55	22	2177

Algoritmo VNS.

Tamaño (n)	Kmax	Valor objetivo	Iteraciones	Tiempo (Milis)
10	3	14	17	7
15	3	7,5	12	15
15	4	9,5	13	14
15	5	9,75	13	15
20	3	12,5	14	17
20	4	13,16	8	8
20	5	12,85	7	7
50	3	46,73	14	93
100	3	78,55	10	497
100	4	78,34	14	1218
100	5	78,34	8	474