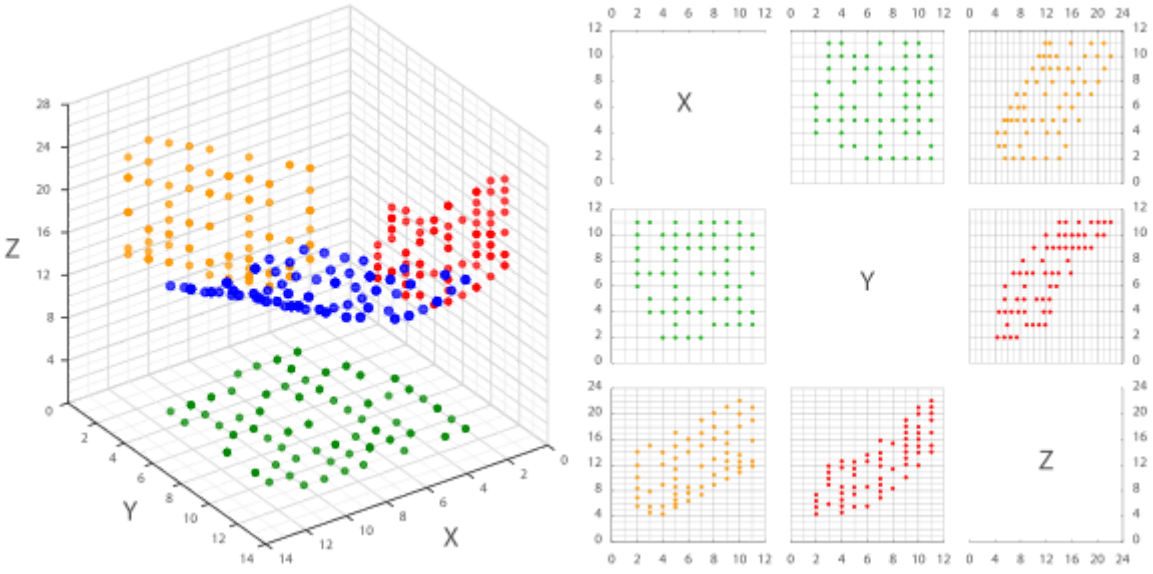
Max Mean Dispersion Problem

Lucas Christian Bodson Lobato

alu0101111254



Índice

Índice **2**

Introducción **3**
- Objetivo: 3
- Problema: 3

Build **3**
- Requisitos: 3
- Construcción: 3

Ejecución **3**

Estructura de clases **4**

Algoritmos usados **6**
- Greedy Constructivo: 6
  - Explicación: 6
  - Resultados: 6
- Greedy Destructivo: 6
  - Explicación: 6
  - Resultados: 6
- GRASP: 6
  - Explicación: 6
  - Resultados: 6
- Multi-arranque: 6
  - Explicación: 6
  - Resultados: 6
- VNS: 6
  - Explicación: 6
  - Resultados: 6
- Comparación: 7
  - Tiempos: 7
  - Resultados: 7

Referencias **10**
- Remotas: 10
- Locales: 10

Introducción

Objetivo:

El objetivo de esta práctica es realizar un estudio sobre el Max Mean Dispersion Problem utilizando los algoritmos vistos en clase, y comparar los resultados obtenidos con estos.

Para llegar a este fin se ha desarrollado un programa en C++ utilizando un patrón estrategia implementando todos los algoritmos a estudiar.

Problema:

El problema a estudiar se trata de obtener un conjunto de nodos con una dispersión media entre ellos lo mayor posible, es decir, maximizar la función Md(Solución) tal que:

Md(solucion) = Sum(i,j) / |Solucion| tal que i y j pertenecen a solución.

El objetivo de esto es encontrar un conjunto de datos con la mayor distancia entre ellos posible, lo cual tiene utilidad a la hora de hallar datos con diversidad entre ellos, para evitar datos similares entre sí, y evitar redundancia en otros estudios.

Especificaciones del sistema

Nombre del dispositivo	DESKTOP-4ORJOOA
Procesador	Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz 2.60 GHz
RAM instalada	16,0 GB (15,9 GB usable)
Identificador de dispositivo	EB11B836-F8E3-4266-8D7D-51B108F9E131
Id. del producto	00326-10000-00000-AA493
Tipo de sistema	Sistema operativo de 64 bits, procesador basado en x64

La compilación y ejecución se realizó utilizando el WSL (Windows Subsystem for Linux) como consola.

Build

Requisitos:

Para construir el programa es necesario tener instalado g++, y preferiblemente un ordenador con un sistema linux.

Construcción:

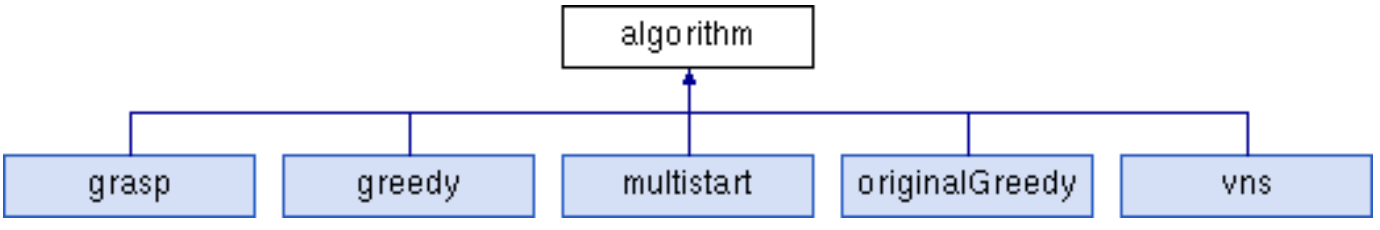
Para construir el programa basta con compilar todos los ficheros .cpp contenidos en src y el fichero main.cpp contenido en build con un compilador de c++, o teniendo instalado g++ y make, ejecutar make en la carpeta build del proyecto.

Ejecución

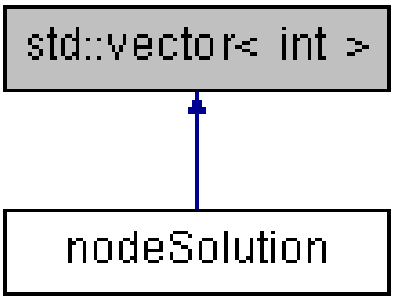
Para ejecutar el programa entre en el directorio bin y ejecute el ejecutable contenido dentro de estar en un sistema linux, o ejecute el programa generado en caso de no compilar usando g++.
Al ejecutar el programa se le preguntarán los inputs que desea usar, y se le pedirán paso a paso las opciones que desea.

Estructura de clases

La estructura de Clases usada en esta práctica está basada en el patrón estrategia, es decir, existe una clase padre virtual algoritmo de la que heredan todos los algoritmos a estudiar, y una clase interfaz que nos sirve para interactuar con estas clases y sus funcionalidades.



Además de esto existe la clase nodeSolution donde se almacena el vector de nodos solución.



Y una clase distanceMatrix donde almacenaremos el contenido del problema en una matriz de distancias, que obtendremos a partir de una clase fileHandler que se encarga de la lectura de ficheros.

Algoritmos usados

Greedy Constructivo:

Documentación

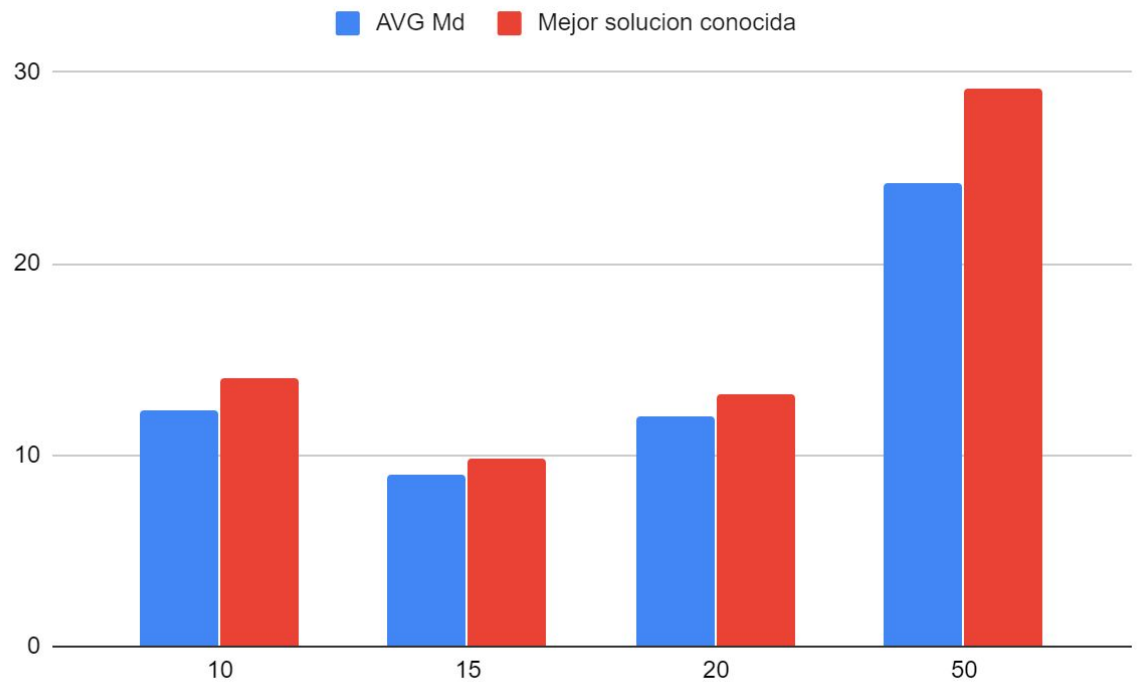
Explicación:

Este algoritmo está basado en el pseudocódigo proporcionado para la realización de esta práctica, su funcionamiento general es tras generar una solución general seleccionando al azar una arista entre todas las aristas con valor máximo del problema buscamos el nodo que más mejora la solución al añadirse a esta, y seleccionamos uno al azar en caso de haber múltiples, y repetiremos esto hasta no encontrar mejora.

Resultados:

Al ejecutar el algoritmo con los datos proporcionados vemos que solo en pocos casos llega a la mejor solución obtenida en el caso del fichero con 10 nodos, y en el caso de 15 nodos nunca llega a esta, empeorando aún más con problemas de mayor tamaño, sin embargo este algoritmo tiene la ventaja de ser el más rápido de todos los estudiados, sacrificando resultados por velocidad, con lo cual puede ser útil para ejecutarse de forma consecutiva con datos iniciales distintos para algoritmos multi-arranque.

GREEDY CONSTRUCTIVO					
Problema	Nodos	Mejor solución conocida	Ejecución	AVG Md	AVG Run Time
max-mean-div-10.txt	10	14	total = 10	12,35716	0,44ms
max-mean-div-15.txt	15	9,83333	total = 10	8,975	0,31ms
max-mean-div-20.txt	20	13,1674	total = 10	12,0538	0,90ms
max-mean-div-50.txt	50	29,1333	total = 10	24,20389	4,58ms



Greedy Destructivo:

Documentación

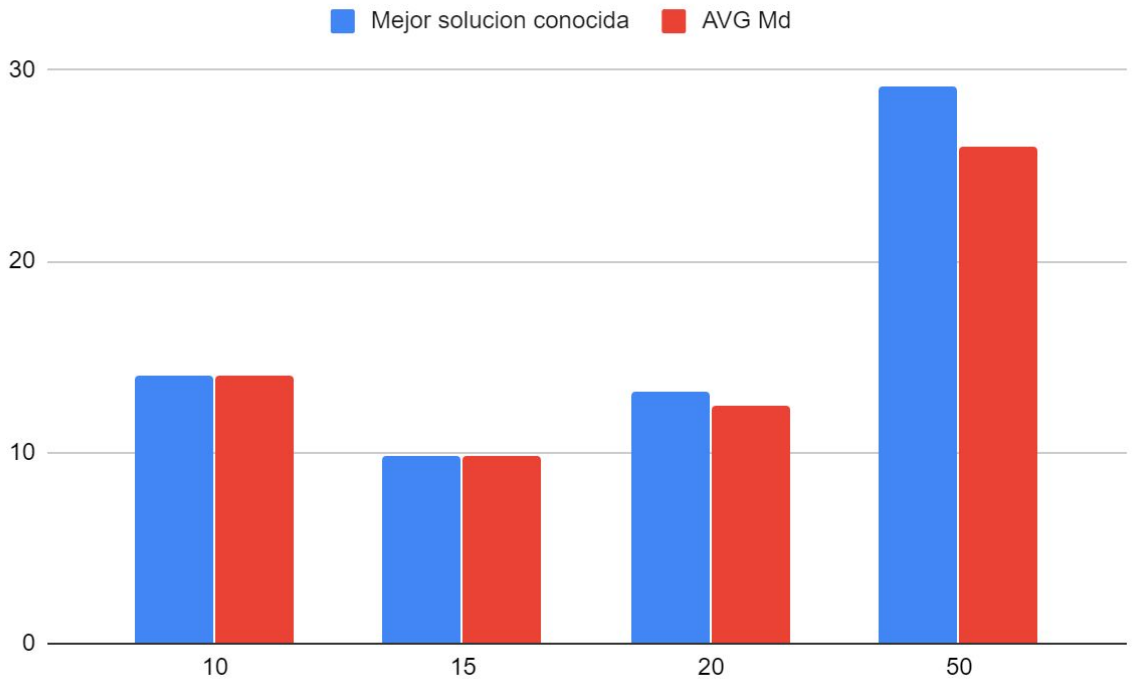
Explicación:

Este algoritmo se basa en la inversión de el objetivo de el primer algoritmo greedy, es decir en lugar de empezar con un vector pequeño y añadiendo los mejores nodos en cada iteración, empezamos con un vector lleno y eliminamos el peor nodo en cada iteración, es decir buscamos el nodo que al eliminarse mejore más la solución y lo eliminamos de ella, en caso de haber varios, como en el caso del primer greedy, seleccionamos uno al azar para eliminar.

Resultados:

Los resultados obtenidos con este algoritmo siguen sin ser perfectos en comparación a la mejor solución conocida, pero aun así existe una gran mejoría en comparación al algoritmo greedy constructivo que se nos proporcionó para el estudio, aunque el tiempo de computación también comienza a ser mayor debido a que por lo general este algoritmo tiene que comprobar más nodos.

GREEDY DESTRUCTIVO					
Problema	**Nodos**	**Mejor solución conocida**	**Ejecución**	**AVG Md**	**AVG Run Time**
max-mean-div-10.txt	**10**	**14**	**total = 10**	**14**	**0,32ms**
max-mean-div-15.txt	**15**	**9,83333**	**total = 10**	**9,83333**	**0,90ms**
max-mean-div-20.txt	**20**	**13,1674**	**total = 10**	**12,42858**	**1,42ms**
max-mean-div-50.txt	**50**	**29,1333**	**total = 10**	**25,9615**	**9,68ms**



GRASP:

[Documentación]

Explicación:

Este algoritmo se basa en dos grandes fases, la fase de construcción, y la búsqueda local, la búsqueda local es simplemente una combinación de los dos algoritmos greedy, en la cual itera hasta no encontrar ninguna mejora en ambos algoritmos, mientras que la fase de construcción se basa en construir un RCL, o Restricted Candidate List, es decir una lista de nodos con los mejores valores de mejora (ojo, no solo los máximos) y seleccionar un nodo aleatorio entre ellos, y repitiendo la generación de RLC con la nueva solución, repitiendo estos pasos hasta llegar a tener una solución de un tamaño aleatorio definido previamente.

El algoritmo en sí funciona iterando ejecuciones de la fase de construcción, y la búsqueda local sobre la solución resultante buscando una solución mejora a la ya encontrada hasta que se llegue a una condición de parada., tras la cual se devolverá la mejor solución encontrada hasta el momento.
En el caso de este estudio se optó por usar dos condiciones de parada, número máximo de ejecuciones y número máximo de ejecuciones sin mejora.

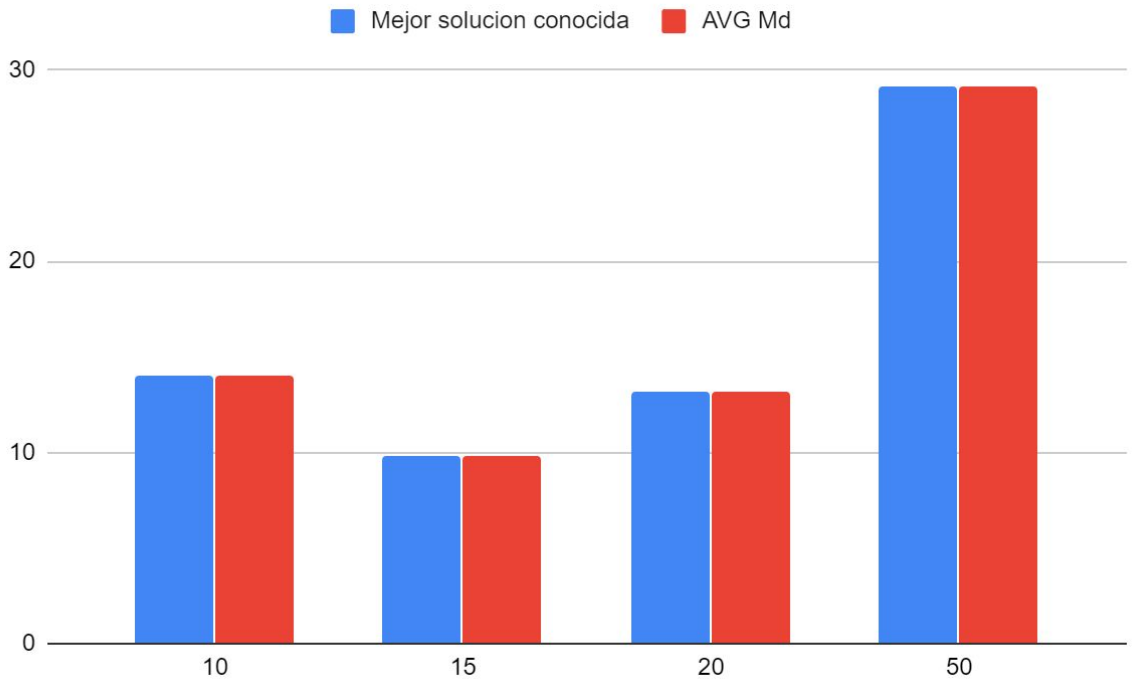
Para generar el RCL en este caso se ordena en un vector los nodos por solución y se devuelve el porcentaje alfa mejor de este vector. El hecho de que se ordena el vector es una de las causas de que este algoritmo tenga el peor tiempo de ejecución.

Resultados:

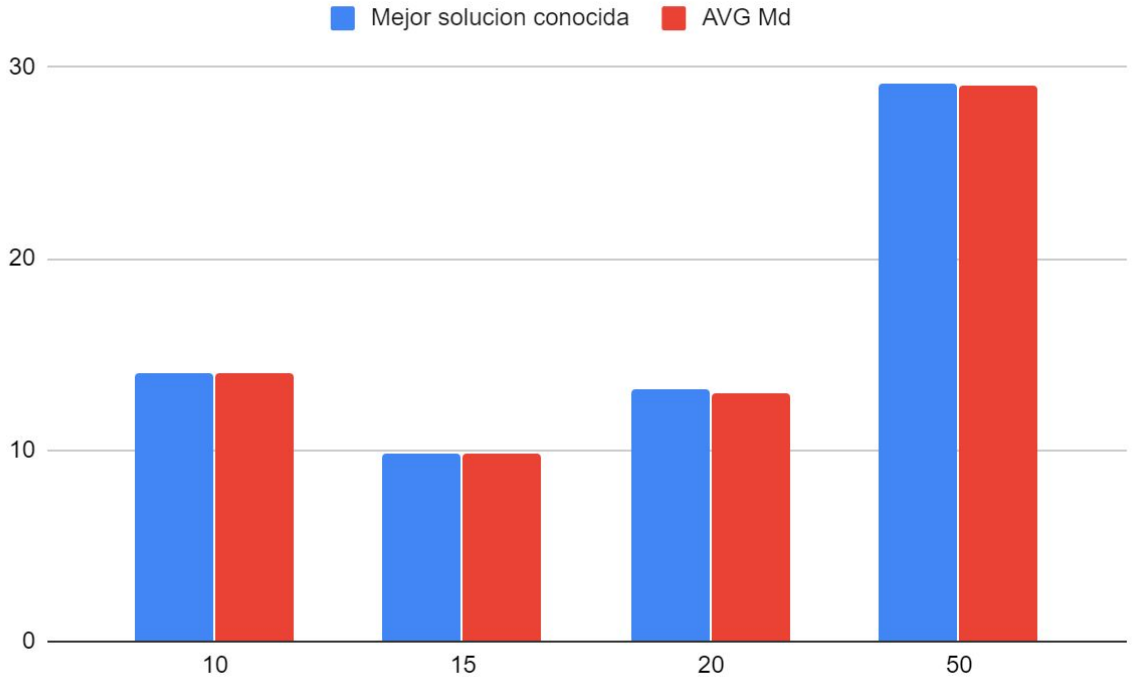
Como se puede observar en grasp los resultados son en la mayoría de los casos los mejores conocidos, pero en problemas grandes utilizando 100 iteraciones máximas, y 10 iteraciones sin mejora comenzamos a ver que no siempre coinciden con el mejor caso conocido, aunque es una diferencia ligera.

Por otra parte los tiempos de ejecución para este algoritmos son de media los más altos entre los algoritmos estudiados.

GRASP									
Problema	Nodos	Mejor solución conocida	Modo	Ejecución	Iteraciones máximas	Iteraciones sin mejora máximas	Alpha	AVG Md	VG run tim
max-mean-div-10.txt	10	14,00	Greedy	total = 10	1000	100	0.2	14	7,70ms
max-mean-div-15.txt	15	9,83	Greedy	total = 10	1000	100	0.2	9,83333	16,90ms
max-mean-div-20.txt	20	13,17	Greedy	total = 10	1000	100	0.2	13,1667	39,60ms
max-mean-div-50.txt	50	29,13	Greedy	total = 10	1000	100	0.2	29,1333	392,50ms



1000 iteraciones máximas, 100 iteraciones sin mejora, alfa 0.2, greedy



100 iteraciones máximas, 10 iteraciones sin mejora, alfa = 0.2, greedy

Multi-arranque:

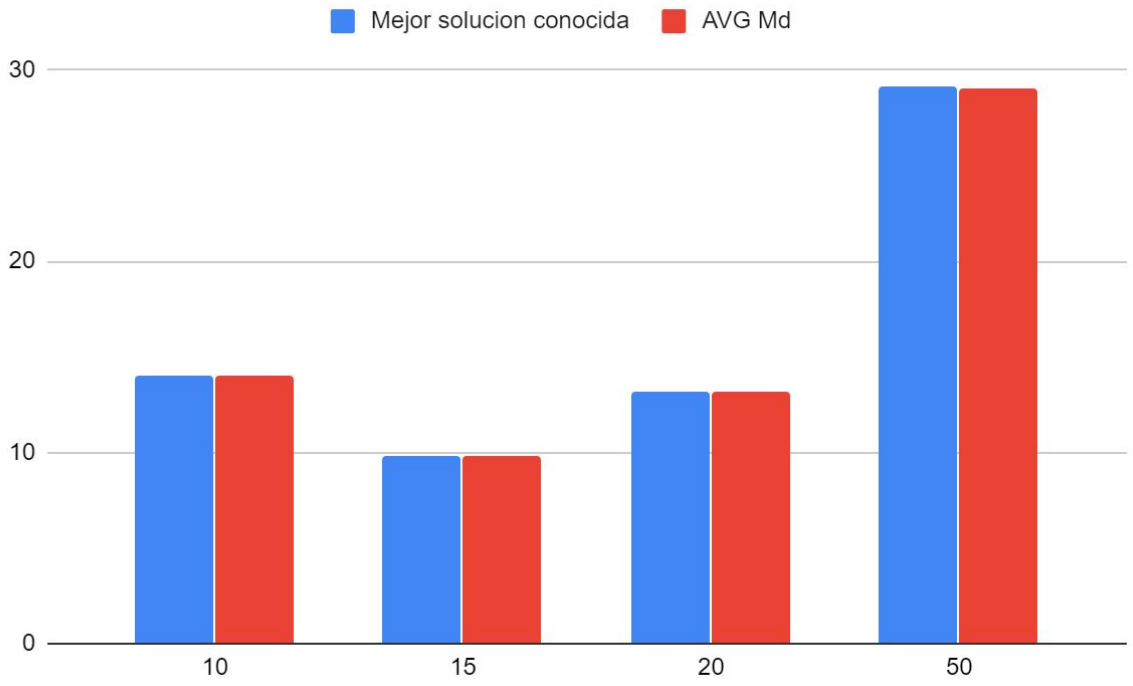
Documentación.

Explicación:

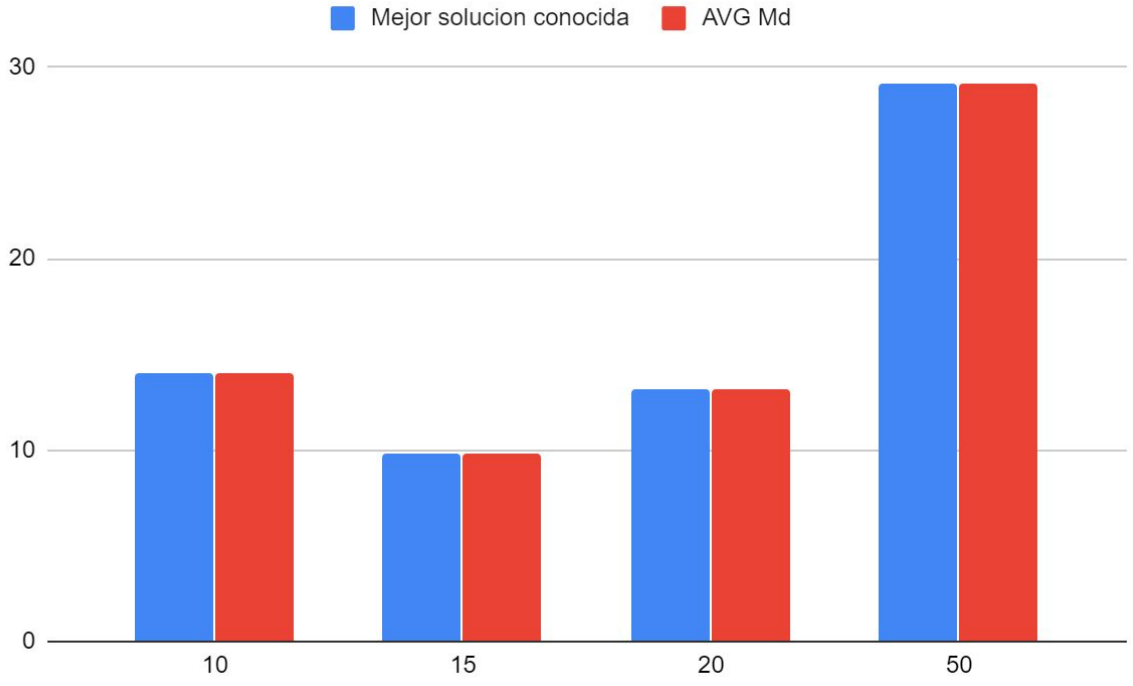
El algoritmo multiarranque tiene la misma estructura general que el algoritmo GRASP, ya que este técnicamente es un método multiarranque, por lo cual su código es muy similar, sin embargo varía en la fase de construcción, en la que en lugar de generar un vector utilizando RCL, se genera de forma completamente aleatoria, dándole más responsabilidad de encontrar una buena solución a la búsqueda local que a la fase de construcción.

Resultados:

MULTI-ARRANQUE								
Problema	**Nodos**	**Mejor solución conocida**	**Modo**	**Ejecución**	**Iteraciones máximas**	**Iteraciones sin mejora máximas**	**AVG Md**	**AVG run time**
max-mean -div-10.txt	**10**	**14**	**Greedy**	**total = 10**	**1000**	**100**	**14**	**3,90ms**
max-mean -div-15.txt	**15**	**9,83333**	**Greedy**	**total = 10**	**1000**	**100**	**9,83333**	**7,00ms**
max-mean -div-20.txt	**20**	**13,1674**	**Greedy**	**total = 10**	**1000**	**100**	**13,1667**	**13,80ms**
max-mean -div-50.txt	**50**	**29,1333**	**Greedy**	**total = 10**	**1000**	**100**	**29,1333**	**110,70ms**



100 iteraciones máximas, 10 iteraciones sin mejora, greedy.



1000 iteraciones máximas, 100 iteraciones sin mejora, greedy

VNS:

Documentación

Explicación:

El algoritmo vns, comparte el elemento de la búsqueda local con el multi-arranque y con el GRASP, pero no comparte la fase de construcción con ellos, ya que se basa en la generación de soluciones por entornos, durante cada iteración se entra un bucle while condicionado por Kmax, siendo Kmax el número de entornos máximos al que se puede acceder, durante cada iteración de este bucle se genera una solución aleatoria en el entorno K de la solución actual, este proceso se denomina shake, y en el caso de este estudio utiliza el tamaño de la solución como entorno, de forma que cualquier solución con un tamaño mayor o menor que el actual por 1 estaría en el entorno k = 1 de la solución actual.

Tras realizar el shake se hace una búsqueda local, y si se encuentra una solución mejor se resetea el valor de k y se sustituye la solución actual por la mejor encontrada, en caso contrario se aumenta el valor de k por 1 y se vuelve a iterar, repitiendo esto hasta que k valga Kmax.

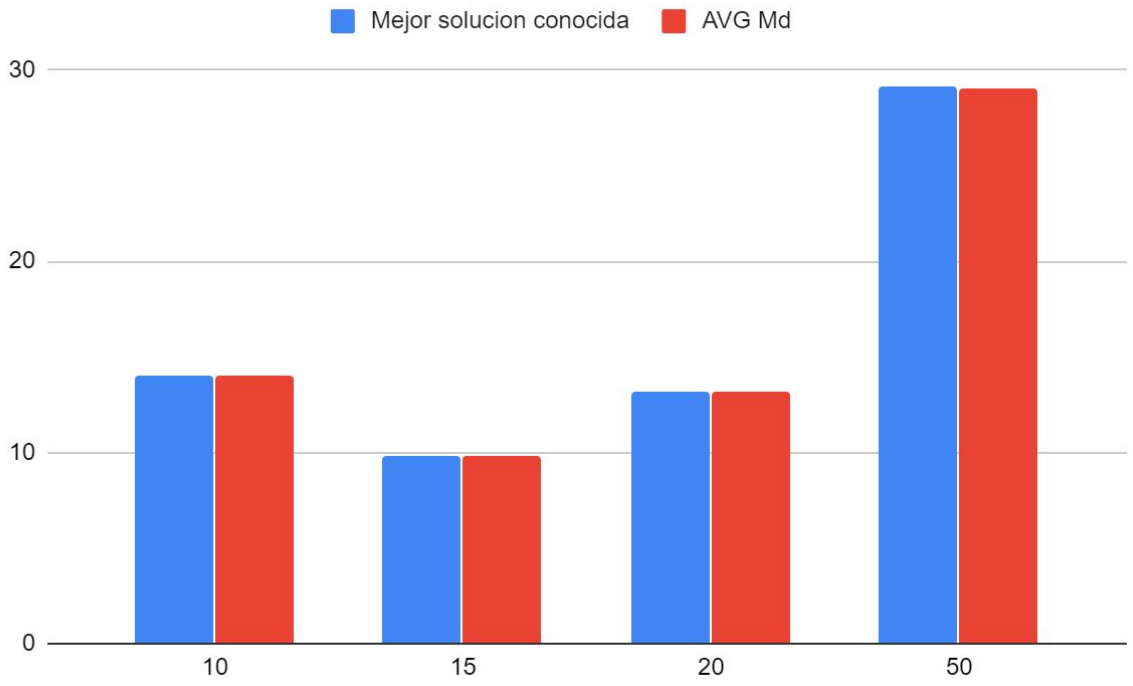
Este proceso se repetirá hasta llegar a las condiciones de parada.

Resultados:

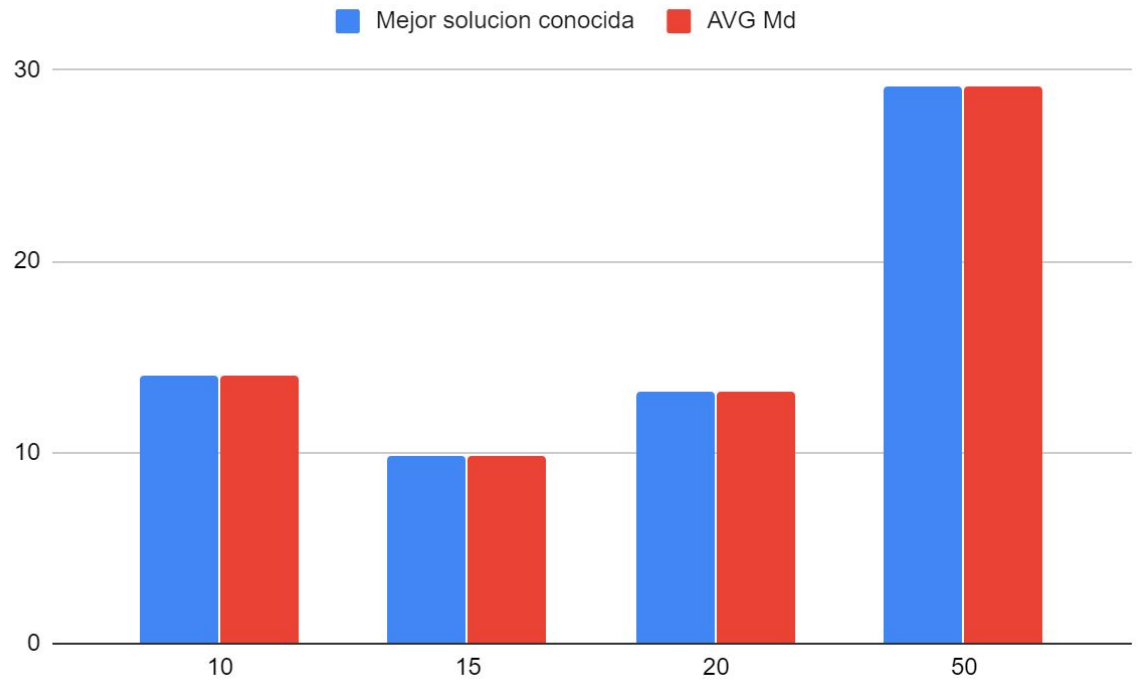
Los resultados obtenidos por el vns son los mejores en general, ligeramente por encima del método multiarranque, tanto en tiempo, como en resultados, ya que obtiene la mejor solución conocida en la gran mayoría de los casos, la única excepción siendo en el caso de 50 nodos, donde tras 10 ejecuciones con 100 iteraciones máximas y 10 iteraciones sin mejora la media obtenida fue de 29.06, siendo el máximo conocido un 29.13.
En cuanto a tiempo, su rendimiento es el mejor entre los algoritmos multiarranque.

VNS									
Problema	**Nodos**	**Mejor solución conocida**	**Modo**	**Ejecución**	**Iteraciones máximas**	**Iteraciones sin mejora máximas**	**Kmax**	**AVG Md**	**AVG run time**
max-mean-div-10.txt	**10**	**14**	**Greedy**	**total = 10**	**1000**	**100**	**2**	**14**	**3,40ms**
max-mean-div-15.txt	**15**	**9,83333**	**Greedy**	**total = 10**	**1000**	**100**	**2**	**9,83333**	**5,10ms**
max-mean-div-20.txt	**20**	**13,1674**	**Greedy**	**total = 10**	**1000**	**100**	**2**	**13,1667**	**7,50ms**

max-mean-div-50.txt	50	29,1333	Greedy	total = 10	1000	100	2	29,1333	60,10ms



100 iteraciones máximas, 10 iteraciones sin mejora, kmax = 2, greedy

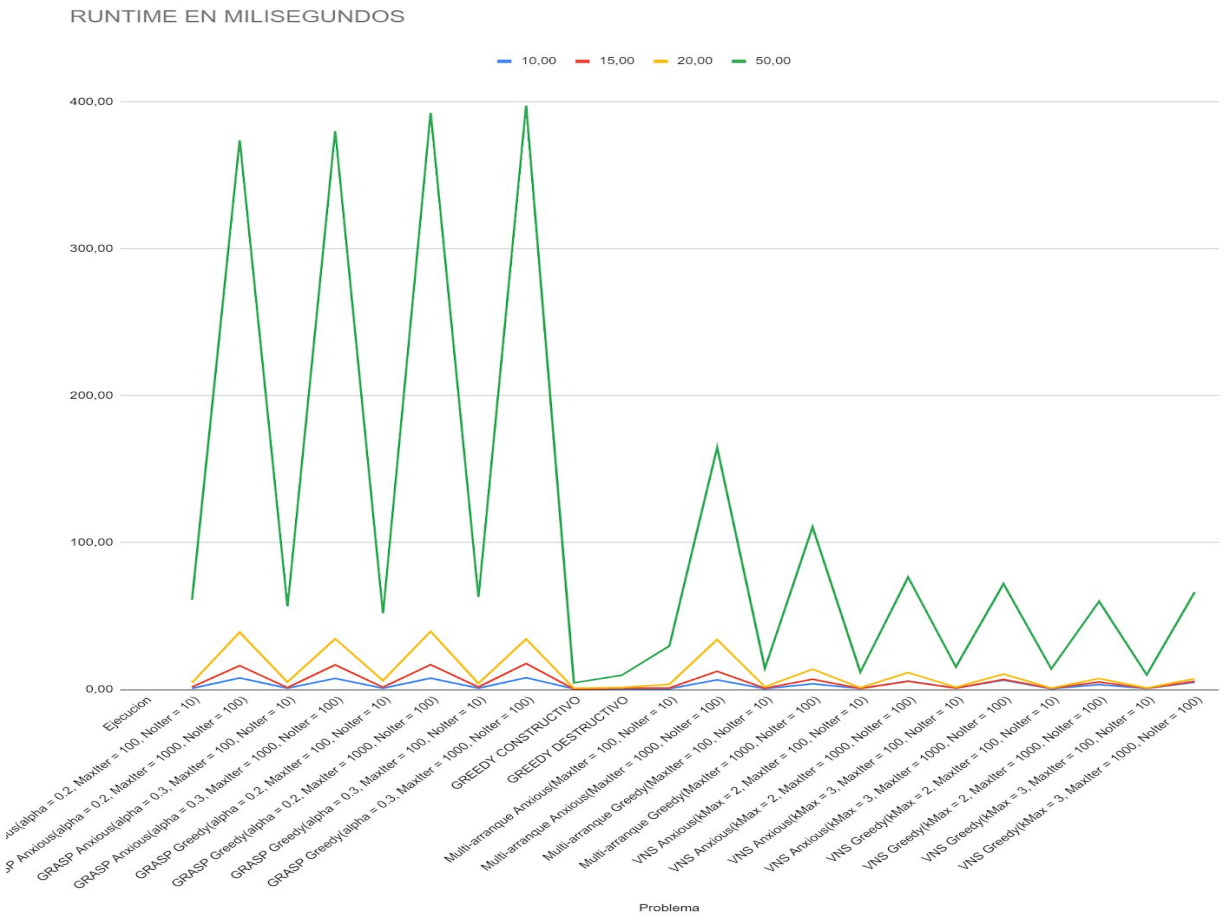


1000 iteraciones máximas, 100 iteraciones sin mejora, kmax = 2, greedy

Conclusiones:

Tiempos:

La variable que más cambia según el algoritmo usado durante el estudio fue principalmente el tiempo, ya que para observar *grandes* diferencias entre los resultados obtenidos sería necesario realizar un estudio exhaustivo, en un entorno de pruebas más “estéril”, y con un conjunto de datos más grande y variado, pero con los resultados que hemos obtenido podemos hacer predicciones y estimaciones, en este caso vamos a usar los resultados con 50 nodos, ya que es con estos con los que más se notaba la diferencia en tiempos de los algoritmos, como se puede ver en la gráfica siguiente.



Como se puede observar los algoritmos que menos tiempo necesitaron para terminar de ejecutarse fueron los algoritmos greedy, siendo el constructivo casi dos veces más rápido que el destructivo. Tras estos el algoritmo más rápido es el VNS que tanto con un número alto de iteraciones como con un número alto parece obtener mejores tiempos que el grasp y el multiarranque, aunque con un kmax muy elevado esto podría cambiar.
Finalmente el multiarranque es más rápido que el GRASP, ya que genera soluciones completamente aleatorias en la fase de construcción, en lugar de generar un RCL del cual seleccionar los nodos para la solución.

Un dato curioso es que la ejecución usando el algoritmo ansioso no mejora el tiempo de ejecución, ya que el tiempo que gana buscando la primera solución con mejora lo pierde teniendo que hacer múltiples cambios en los nodos de la solución antes de llegar al local óptimo, de hecho en el caso de VNS tarda más el caso ansioso que el greedy.

Resultados:

Las diferencias entre resultados, aunque menores que las diferencias entre tiempos, existen, y al igual que los tiempos son más fáciles de observar en el problema de 50 nodos, siendo los resultados, en orden de mejores a peores resultados los siguientes:

1. VNS: obtiene de forma consistente los mejores resultados, tardando menos tiempo en obtenerlos que los otros dos algoritmos multi-arranque.
2. Multi-arranque: aunque sus resultados en el fichero de 50 nodos fueron ligeramente peores, es mucho más eficiente en términos de tiempo, y obtuvo resultados perfectos en el fichero de 20 nodos.
3. GRASP: Muy similar a VNS y Multi-arranque, pero falló en el fichero de 20 nodos, y además de esto es el algoritmo que más tiempo de ejecución requiere de media.
4. Greedy destructivo: Como ya se ha indicado previamente este consigue resultados superiores al greedy constructivo, sacrificando a cambio eficiencia de tiempo.
5. Greedy constructivo, este algoritmo al igual que el greedy destructivo es usado para las búsquedas locales de los algoritmos GRASP, VNS y Multi-arranque, pero sus resultados de forma separada no son tan buenos.

Referencias

Remotas:

- Tablas con información de ejecución (Contiene información mucho más extensa)
- Repositorio con el código
- Documentación del código

Locales:

- /Bin -> ejecutable
- /src -> ficheros.cpp
- /Documentacion -> Documentacion generada por doxygen
- /include -> ficheros hpp
- /Result data -> tablas de resultados
- /build -> makefile y fichero main.cpp