

# Metaheurística

---

## Práctica 5 Multidimensional 2-way Number Partitioning Problem

Adrián Lopez Ortíz  
José Jesús Torronteras Hernández  
Juan José Méndez Torrero

Grupo: C

Universidad de Córdoba

30 de mayo de 2019

### Resumen

In the present document we will show how to solve the *Multidimensional 2-way Number Partitioning Problem* (M2-NPP) using C++ language. Furthermore, we will explain what is the problem about and how we have solved it. In addition, we will explain all the metaheuristics that we have used in order to solve the problem. At the end of the document, we will explain our conclusions about the results achieved.

# Índice

<b>1. Introducción</b>	<b>3</b>
<b>2. Instancias</b>	<b>6</b>
<b>3. Clases</b>	<b>7</b>
<b>4. Metaheurísticas</b>	<b>8</b>
4.1. Búsqueda aleatoria . . . . .	8
4.2. Búsqueda Local - Escalada simple . . . . .	9
4.3. Búsqueda Local - Escalada por máxima pendiente . . . . .	11
4.4. GRASP . . . . .	12
4.5. Enfriamiento simulado . . . . .	14
<b>5. Análisis de los resultados</b>	<b>15</b>
<b>6. Conclusiones</b>	<b>19</b>
<b>7. Grado de implicación</b>	<b>19</b>

## 1. Introducción

Para esta práctica vamos a hacer uso de las metaheurísticas aprendidas a lo largo del curso para resolver el problema elegido. Este problema es *Multidimensional 2-way Number Partitioning Problem*(M2-NPP), el cual consisten en, dado  $X$  vectores de dimensión  $D$ , conseguir una distribución en dos conjuntos de tal manera que la máxima diferencia entre estos dos conjuntos sea mínima. Para ello utilizaremos las siguientes metaheurísticas:

- Búsqueda Aleatoria.
- Búsqueda Local - Escalada simple.
- Búsqueda Local - Escalada por máxima pendiente.
- Greedy.

Para una mejor comprensión del problema en cuestión, hemos decidido realizar una explicación a través de imágenes. Como vemos en la Figura 1, el conjunto inicial cuenta con  $X$  vectores de dimensión  $D$ .

5	7	4	3	9	8	5	1	4	2
3	6	5	4	7	8	6	5	9	8
7	0	6	5	3	0	8	5	6	4
3	7	8	9	6	5	3	8	7	5
6	4	3	0	5	3	5	2	8	5
6	4	1	3	2	4	0	8	6	9
6	4	7	6	5	4	9	7	8	3
4	5	2	9	6	7	4	0	1	2
3	5	4	6	7	0	8	9	7	5
5	5	6	2	4	3	5	9	6	8
7	7	6	4	1	2	3	5	6	7
4	1	3	6	9	5	1	4	7	2
5	6	3	2	1	4	7	8	9	6
2	3	1	7	5	8	9	5	1	4
7	5	6	5	2	5	4	7	8	6

Figura 1: Conjunto inicial

Una vez tenemos el conjunto inicial, el algoritmo consiste en dividir ese conjunto en dos conjuntos diferentes, y una vez dividido, calcular la suma de todos los vectores de cada conjunto por separado. Este paso se puede apreciar en las Figuras 2 y 3.


5	5	6	2	4	3	5	9	6	8	SET 1
7	7	6	4	1	2	3	5	6	7	
4	1	3	6	9	5	1	4	7	2	
5	6	3	2	1	4	7	8	9	6	
2	3	1	7	5	8	9	5	1	4	
7	5	6	5	2	5	4	7	8	6	
										
30	27	25	26	22	27	29	38	37	33	SUM1

Figura 2: Primer conjunto con la suma calculada


5	7	4	3	9	8	5	1	4	2	SET 2
3	6	5	4	7	8	6	5	9	8	
7	0	6	5	3	0	8	5	6	4	
3	7	8	9	6	5	3	8	7	5	
6	4	3	0	5	3	5	2	8	5	
6	4	1	3	2	4	0	8	6	9	
6	4	7	6	5	4	9	7	8	3	
4	5	2	9	6	7	4	0	1	2	
3	5	4	6	7	0	8	9	7	5	
										
37	42	40	45	42	39	48	45	48	43	SUM2

Figura 3: Segundo conjunto con la suma calculada

Una vez dividido el conjunto inicial en dos conjuntos diferentes y calculada su suma, pasamos a calcular la diferencia entre estos dos, de tal manera que buscaremos el máximo que haga mínima esa diferencia. Este paso se puede entender mejor fijándose en la Figura 4.

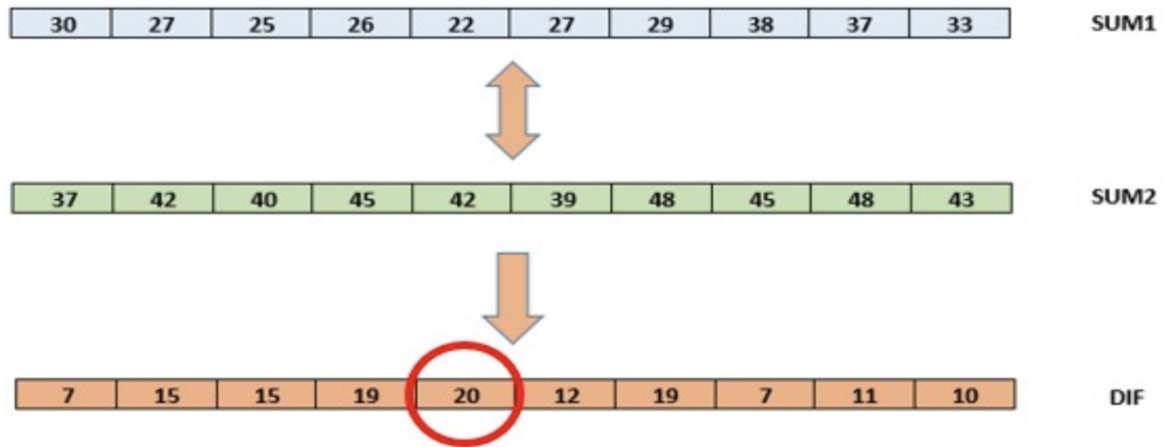


Figura 4: Conjunto diferencia final

Como vemos, nuestro objetivo es dividir el conjunto inicial en dos conjuntos distintos de tal manera que el valor máximo del conjunto diferencia final, sea mínimo. Es decir, para este ejemplo, si encontrásemos una manera de dividir el conjunto inicial de tal manera que el valor máximo del conjunto diferencia final sea menor que 20, podremos decir que esa solución es mejor que la expuesta en el ejemplo.

## 2. Instancias

Para esta práctica hemos creado una serie de archivos en los que guardaremos las instancias utilizadas para resolver el problema seleccionado. Para ello, hemos creado una serie de funciones que nos creará un fichero que incluirá el conjunto de particiones creados de manera aleatoria. Hemos dividido estos conjuntos en los siguientes:

- **Conjunto pequeño:** Este conjunto cuenta con 400 vectores de tamaño 800.
- **Conjunto mediano:** Este conjunto cuenta con 600 vectores de tamaño 800.
- **Conjunto grande:** Este conjunto cuenta con 1200 vectores de tamaño 400.

El objetivo de esta práctica es aplicar una serie de metaheurísticas a estos conjuntos de vectores y posteriormente analizar los resultados obtenidos y compararlos.

### 3. Clases

Para la realización de esta práctica hemos decidido crear una serie de clases para poder realizar de manera más sencilla la solución del problema seleccionado. Las clases creadas son las siguientes:

- **Clase Partición:** Esta clase contendrá un vector de enteros con un valor aleatorio entre 1 y 10. Además, contendrá dos variables más para controlar el tamaño y el *fitness* de la partición.
- **Clase Sets:** Esta clase es la encargada de contener el vector de Particiones, creando así nuestro conjunto de vectores. Además, contiene dos variables para el tamaño y para calcular la suma de la partición.

Para simular la división del conjunto de particiones inicial, hemos creado una estructura dentro del fichero *Metaheurísticas.hpp* la cual consta de dos variables de tipo Sets y, una variable de tipo entero para controlar el mejor *fitness* obtenido.

Con respecto a la ejecución de las metaheurísticas creadas en los ficheros *Metaheurísticas.cpp* y *Metaheurísticas.hpp*, hemos creado los ficheros *Funciones.cpp* y *Funciones.hpp* para poder ejecutar cada metaheurística desde el programa principal de una manera más sencilla.

## 4. Metaheurísticas

En esta sección hablaremos de las distintas metaheurísticas usadas para este problema y de cómo hemos cambiado la estructura con respecto al problema de la mochila múltiple. Para ello, dividiremos la sección según la metaheurística creadas.

### 4.1. Búsqueda aleatoria

De igual manera que en el problema de la mochila, para este problema elegiremos de manera aleatoria a qué conjunto entra, si al primero o al segundo. Una vez realizada esta división aleatoria, calcularemos la diferencia entre ellos y guardamos el máximo de ese conjunto resultante que será nuestro valor de *fitness*.

Con respecto al código correspondiente a esta metaheurística, hemos creado dos variables de tipo Set en las que almacenaremos aleatoriamente los vectores que leeremos de un fichero. Finalmente, calcularemos la diferencia de los dos conjuntos y la devolveremos para poder mostrar los resultados.

Los resultados tras aplicar esta metaheurística para cada uno de los conjuntos se puede apreciar en la Figura 5.



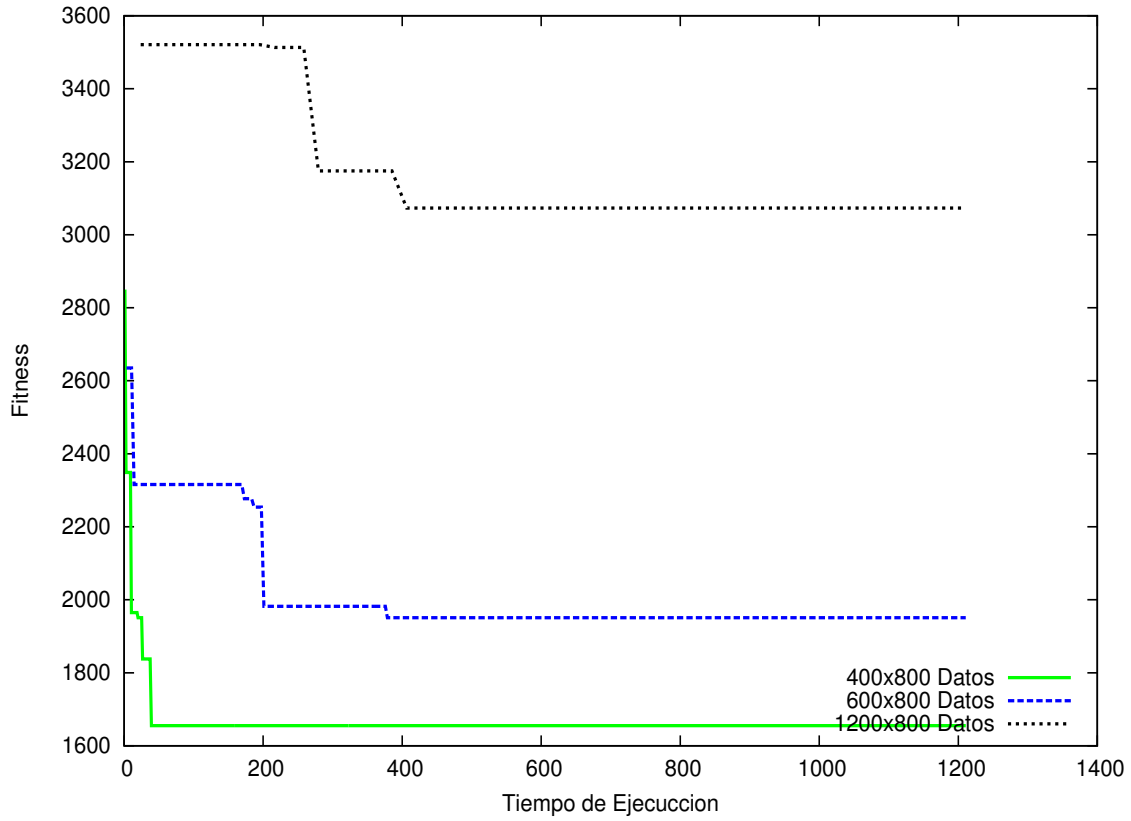


Figura 5: Resultados obtenidos con búsqueda aleatoria

#### 4.2. Búsqueda Local - Escalada simple

Esta metaheurística consiste en, primero, recibir una solución aleatoria a partir de la cual se irá explorando el vecindario de esta solución hasta encontrar una solución que su valor de *fitness* sea mejor que el actual.

Para esta exploración lo que haremos será cambiar una partición(vector) de un conjunto a otro conjunto. Cuando esta exploración encuentra una mejor solución actual, esta solución es devuelta y se comienza esa exploración a partir de ésta última hasta que el algoritmo no encuentre un vecino que mejore a la actual.

Para la realización del código de esta metaheurística, hemos creado dos variables auxiliares para poder guardar los vectores leídos de un

fichero. Estas variables son de tipo Set y entero. Una vez guardado la solución aleatoria inicial, procederemos a realizar la partición para así poder calcular el *fitness* final. Calculado dicho *fitness* la función devolverá ese valor para poder mostrarlo posteriormente en una gráfica.

Los resultados obtenidos con esta metaheurística se pueden apreciar en la Figura 6.

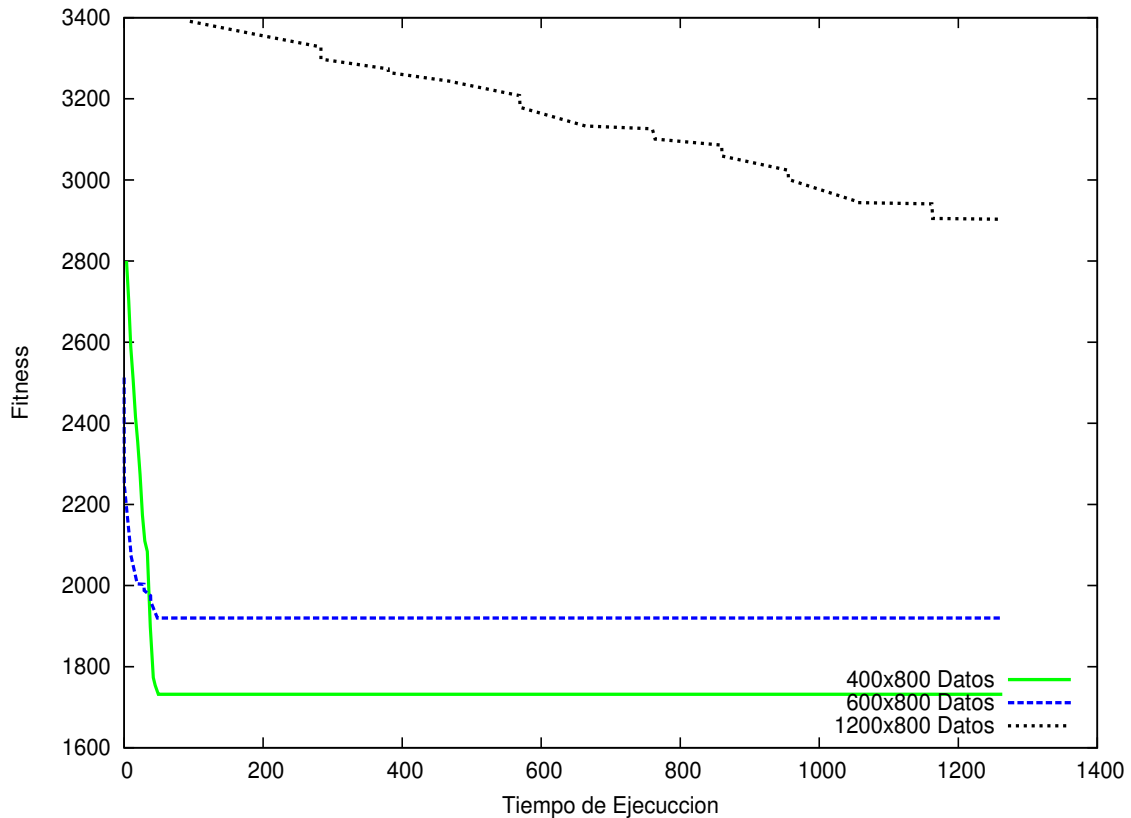


Figura 6: Resultados obtenidos con búsqueda local - escalada simple

### 4.3. Búsqueda Local - Escalada por máxima pendiente

Al igual que en la escalada simple, esta metaheurística recibirá una solución aleatoria al principio, una vez recibida, este algoritmo explorará el vecindario en su totalidad y en el caso de encontrar una solución que mejora a la actual, ésta será devuelta y se procederá a reiniciar esa exploración desde la mejor solución hasta que no se encuentre una mejor.

Este algoritmo suele ser más lento que el anterior ya que el algoritmo tendrá que explorar todo el vecindario antes de devolver una solución. Aunque sea más lento, este algoritmo será capaz de encontrar mejores soluciones que con la escalada simple.

El código de esta metaheurística es muy parecido al de la escala simple, con la diferencia de cómo escogemos la siguiente solución que mejora a la actual. Para ello, añadiremos la condición de que si no ha terminado de explorar todo el vecindario, que el algoritmo siga comparando el valor de *fitness* de los demás vecinos de la solución actual.

Tras aplicar esta metaheurística a cada una de nuestras instancias, hemos conseguido los resultados que se muestran en la Figura 7.

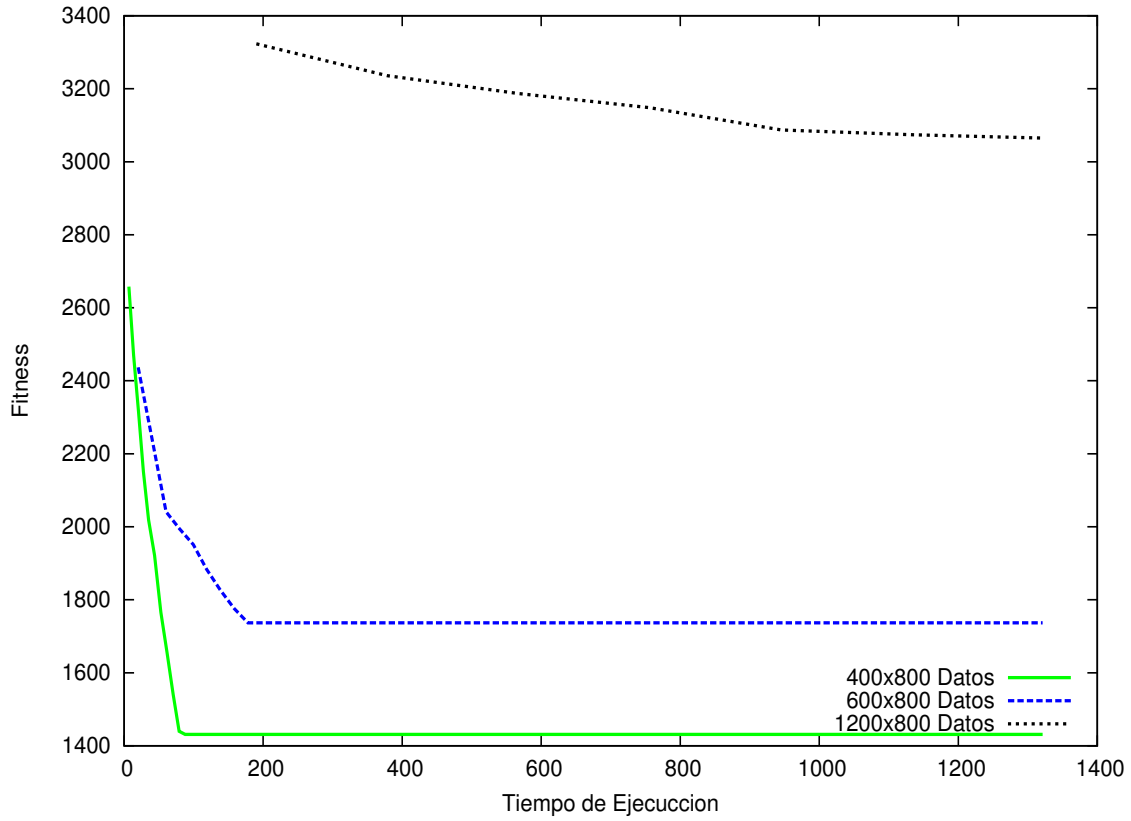


Figura 7: Resultados obtenidos con búsqueda local - escalada por máxima pendiente

#### 4.4. GRASP

El algoritmo *GRASP* recibe una instancia, genera las particiones y las guarda en un vector auxiliar para poder así ordenarlas de mayor a menor seleccionando una columna al azar, creando así nuestro operador de aleatoriedad. Una vez ordenado este vector auxiliar, se añaden las dos primeras particiones a los conjuntos respectivos para inicializarlos.

Recorremos el vector auxiliar de particiones para elegir una partición de las que crearemos otras dos soluciones. Para este recorrido, las dos soluciones anteriores no se cuentan. Al final, de dos soluciones una entrará en un conjunto y la otra en el otro, quedando junto con las dos soluciones iniciales.

Una vez hecho esto, compararemos los fitness de esas dos soluciones

y escogeremos la mejor. Nuestro objetivo es comprobar en que conjunto encaja mejor la partición creada, comparando el valor de *fitness* obtenido en ambos. En general, esta metaheurística tardará mucho menos tiempo que las otras ya que no se recorrerá un vecindario.

Para la codificación de esta metaheurística hemos utilizado las funciones *sort* de la biblioteca *std* para poder así ordenar el vector de manera rápida. Además, creamos un bucle en el que se crearán las nuevas soluciones de la manera anteriormente explicada. Finalmente, esta función devolverá la solución que tenga el menor valor de *fitness*, ya que estamos en un problema de minimización.

Una vez realizado el código, los resultados para cada una de las instancias conseguidos utilizando esta metaheurística se muestran en la Figura 8.

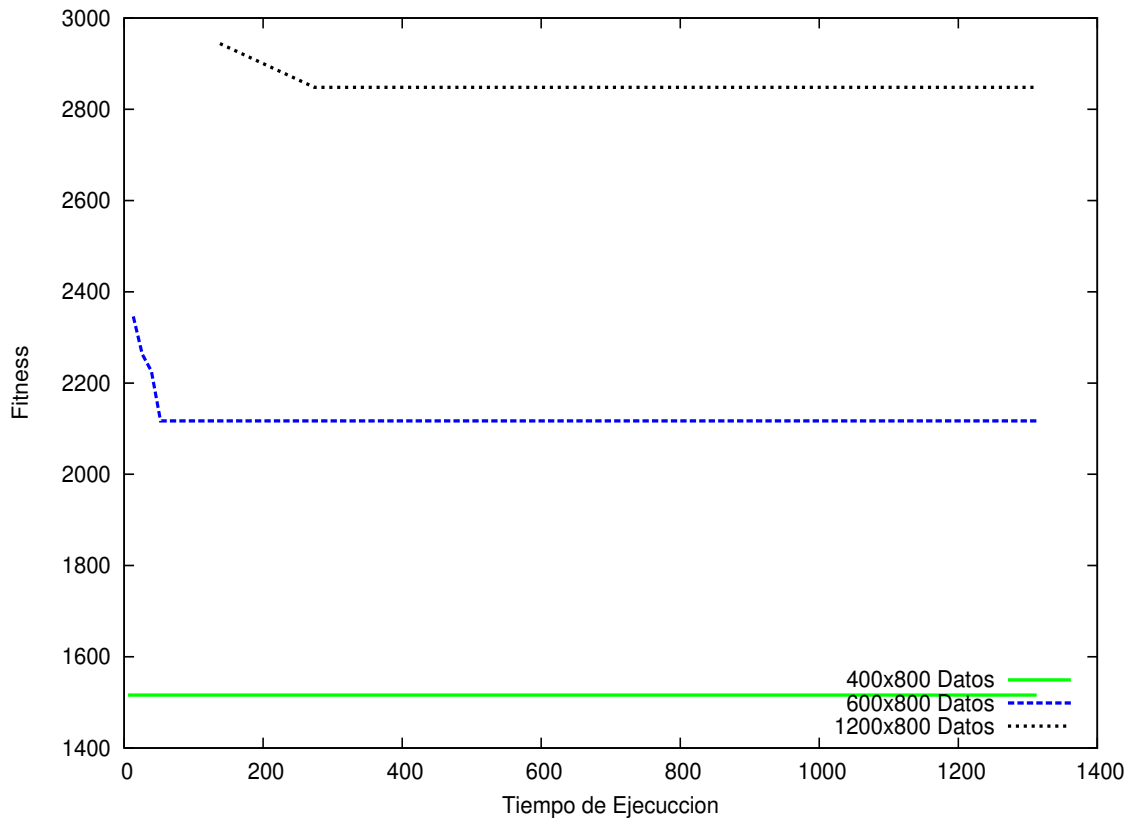


Figura 8: Resultados obtenidos con GRASP

#### 4.5. Enfriamiento simulado

El enfriamiento simulado aplicado para este problema consiste en seleccionar una solución aleatoria inicial, aplicar a esa solución una búsqueda local y seguidamente realizar el algoritmo de enfriamiento. Este algoritmo consiste en ir disminuyendo la *temperatura*, calculada a través de un enfriamiento de tipo logarítmico, para poder aceptar soluciones peores que la actual y así conseguir llegar al óptimo global.

Para este algoritmo hemos creado una estructura para poder guardar las variables de temperatura inicial, temperatura actual, y temperatura final. Además, como en los anteriores algoritmos, hemos creado una función que inicializará todas esas variables y realizará la metaheurística de enfriamiento simulado.

Para realizar el enfriamiento simulado hemos escogido un tamaño aleatorio de vecindad y seguidamente hemos inicializado el vecindario haciendo uso del algoritmo anteriormente creado de búsqueda local. Finalmente, nuestra condición de parada es si la temperatura llega a cero y si no se ha encontrado una mejor solución en un tiempo determinado.

Los resultados que hemos obtenido se pueden observar en la Figura 9.

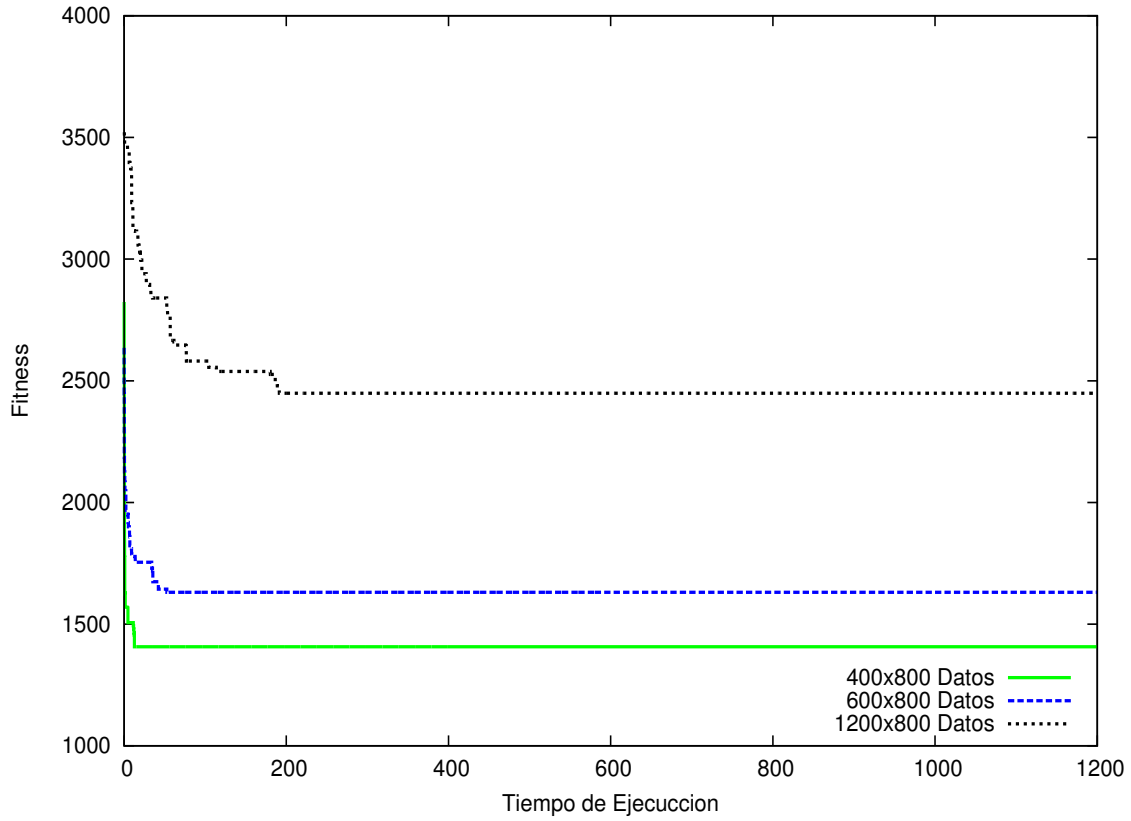


Figura 9: Resultados obtenidos con enfriamiento simulado

## 5. Análisis de los resultados

En esta sección mostraremos los resultados obtenidos con las diferentes metaheurísticas según cada una de los conjuntos de vectores creados. Para ello, dividiremos esta sección según el conjunto de vectores usado.

- **Conjunto pequeño:** Los resultados obtenidos para este conjunto son apreciables en la Figura 10.

Cómo se puede observar en la figura 10, los métodos que mejor fitness obtienen para un conjunto pequeño de instancias, son la búsqueda local con máxima pendiente y la búsqueda con enfriamiento simulado. Las que peor fitness han obtenido han sido la búsqueda local con escalada simple y la metaheurística GRASP que obtiene un fitness similar a la búsqueda aleatoria en este caso.

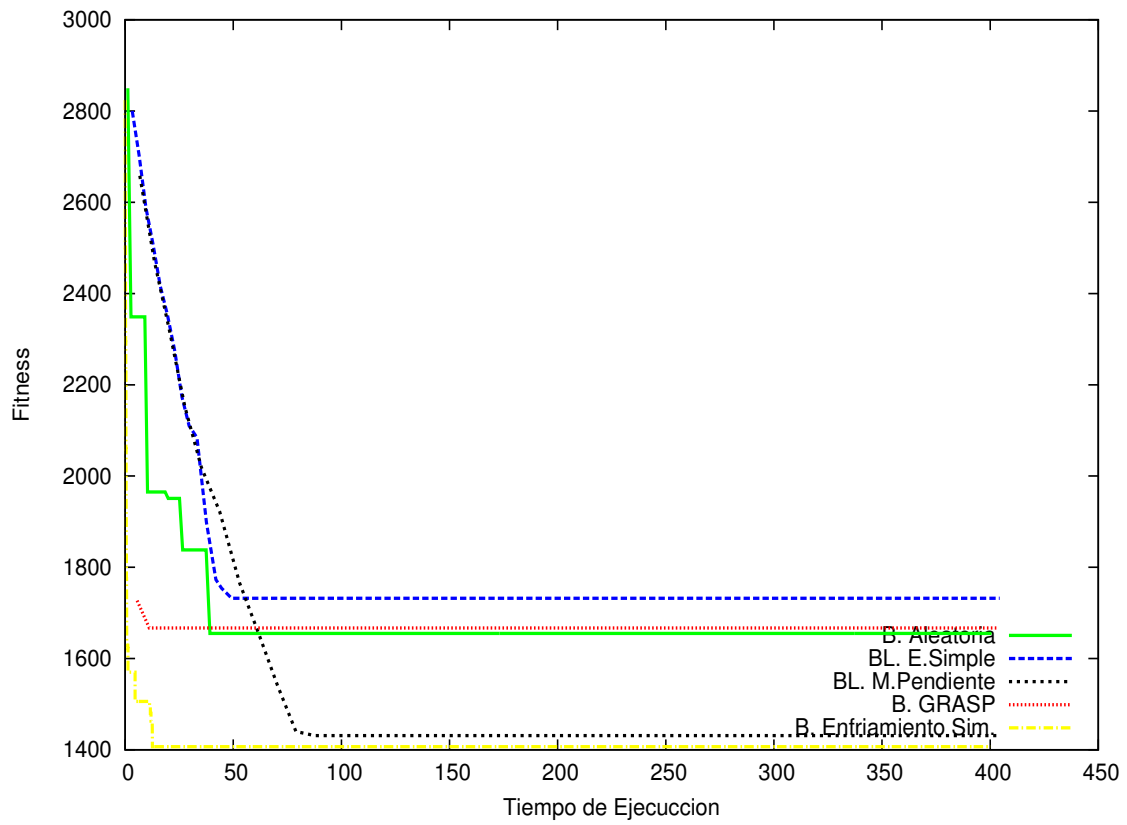


Figura 10: Resultados obtenidos para el conjunto pequeño

- **Conjunto mediano:** Los resultados obtenidos para este conjunto se muestran en la Figura 11.



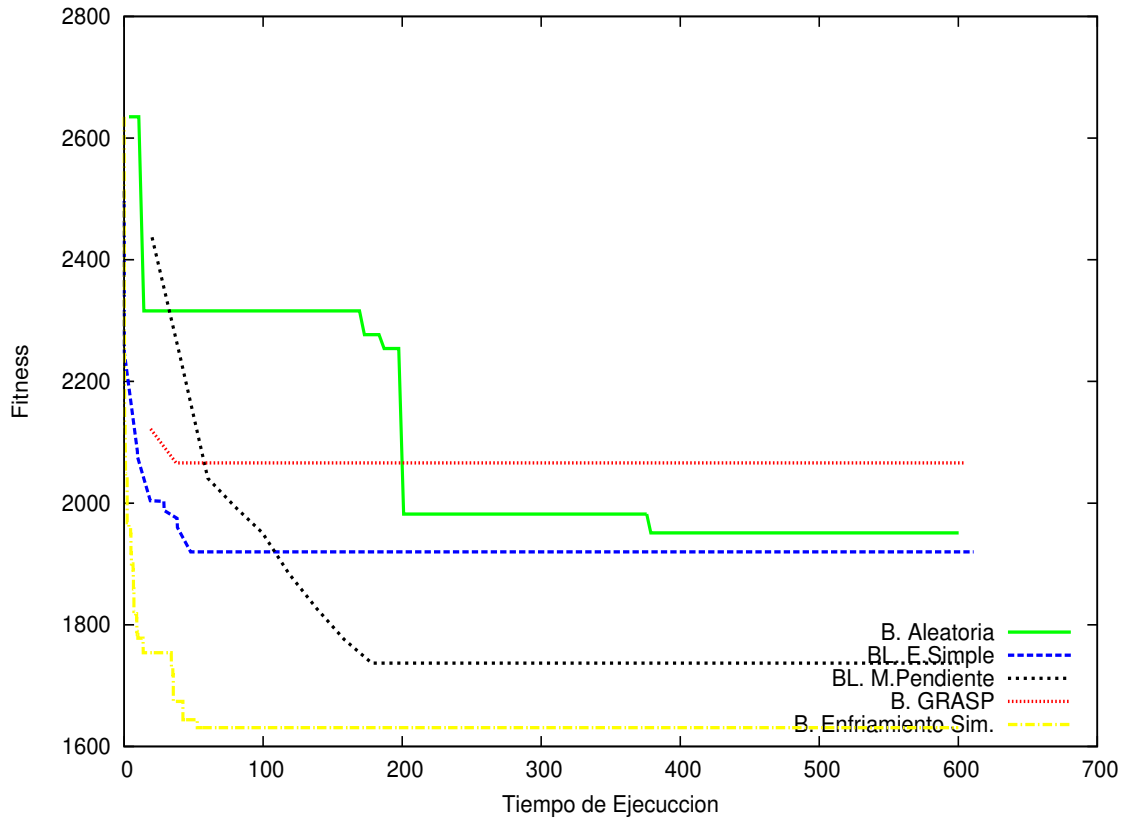


Figura 11: Resultados obtenidos para el conjunto mediano

En esta ocasión, la gráfica de la figura 11, nos muestra que enfriamiento simulado sigue siendo la mejor búsqueda para hallar el mejor fitness, aunque apenas hay diferencia con la búsqueda local de máxima pendiente en cuanto al fitness final, aún así, seguiríamos escogiendo enfriamiento simulado ya que necesita menos tiempo de ejecución para obtener el mejor fitness.

- **Conjunto grande:** Los resultados obtenidos para este conjunto de datos son observables en la Figura 12

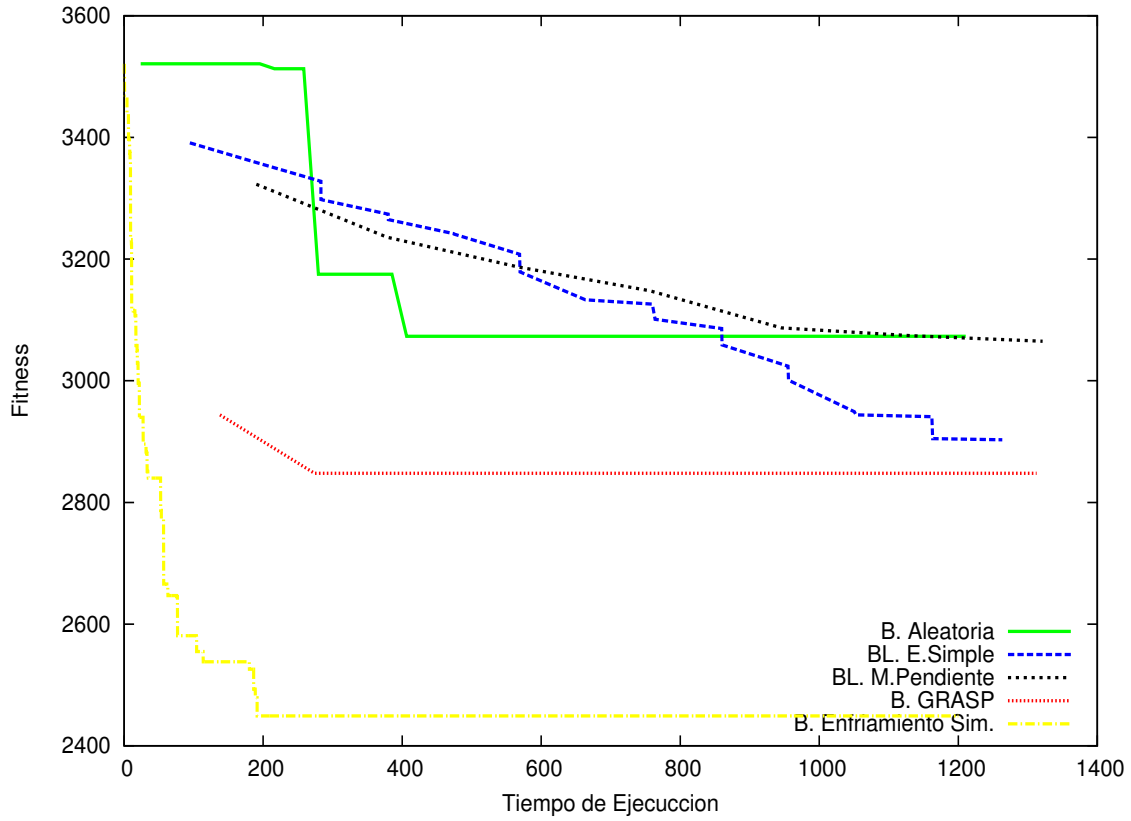


Figura 12: Resultados obtenidos para el conjunto grande

Para un mayor número de instancias se sigue repitiendo la tónica que hemos podido observar en los otros dos casos anteriores, con un número menor de instancias: búsqueda local con escalada simple, búsqueda aleatoria y búsqueda con la metaheurística GRASP obtienen un fitness significativamente peor que las otras dos restantes (búsqueda con enfriamiento simulado y búsqueda local con máxima pendiente). De estas dos, escogeríamos la búsqueda por enfriamiento simulado ya que, pese a alcanzar un fitness similar, esta lo alcanzaría en menor tiempo de ejecución, ya que la búsqueda local con máxima pendiente, en cada búsqueda tiene que analizar el fitness de cada uno de sus vecinos.

## 6. Conclusiones

En esta sección explicaremos las conclusiones generales que hemos obtenido al ver las Figuras 10, 11 y 12. Como hemos podido comprobar, la mejor metaheurística que podemos escoger para nuestro problema, es la de enfriamiento simulado, ya que es la que menor valor de *fitness* tiene. Al ser éste un problema de minimización, vemos que nuestro algoritmo de enfriamiento simulado es capaz de salir de óptimos locales y poder llegar a un óptimo global.

Aunque no ofrece tan buenos resultados como el enfriamiento simulado, la escalada por máxima pendiente ofrece muy buenos resultados para conjuntos pequeños y medianos por lo que también se podría considerar que es una buena metaheurística.

## 7. Grado de implicación

Para la realización de este trabajo nos hemos dividido las tareas según las metaheurísticas que íbamos a realizar, con lo que todos los miembros del grupo han trabajado por igual en la práctica.