Universidad de Valladolid

Facultad de Ciencias

GRADO EN ESTADÍSTICA

ALGORITMOS DE OPTIMIZACIÓN PARA EL PROBLEMA FLOW-SHOP CON NECESIDAD DE RECURSOS ADICIONALES

Pablo Martín de Benito

Tutores:

Juan Camilo Yepes Borrero

Jesús Alberto Tapia García

A María del Carmen Gutiérrez Martín

Resumen

En este Trabajo de Fin de Grado se quiere definir y tratar rigurosamente un problema de ajuste de tareas a máquinas a ejecutar en el mínimo tiempo posible, sujetas a unas restricciones de recursos a través de diferentes algoritmos como heurísticas *greedy* o metaheurísticas GRASP.

Este problema es conocido también como *Flowshop Scheduling Problem* (FSP), un problema NP-Hard, que intentaremos solucionar a través de una búsqueda de la mejor solución factible posible con la ayuda de métodos heurísticos, ya que no podemos obtener una solución óptima debido a la gran complejidad del problema. A lo largo del siguiente trabajo, se proponen diferentes criterios y variantes *greedy* que permiten acercarse a esa mejor solución posible y tomar conclusiones en base a ello.

Además, los experimentos computacionales permiten comparar los resultados obtenidos con diferentes modificaciones del algoritmo asi como con diferentes instancias del problema y se analizarán los resultados obtenidos.

Palabras clave: *Flowshop Scheduling Problem*, heurísticas *greedy*, metaheurísticas GRASP, optimización, algoritmos, *flowshop*, análisis computacional.

Abstract

In this Bachelor's Thesis, we aim to find a feasible solution to the problem of assigning tasks to machines to be executed in the shortest possible time, subject to resource constraints, through different heuristics such as *greedy* heuristics or GRASP metaheuristics.

This problem is also known as the *Flowshop Scheduling Problem* (FSP) and is an NP-Hard problem, where we will try to find the best possible feasible solution since no defined analytical optimal solution exists. Throughout this work, different criteria and *greedy* variants are proposed to approach that best possible solution and draw conclusions based on it.

Additionally, computational experiments will be conducted to compare the results obtained with different modifications of the algorithm, as well as with different problem instances, and the obtained results will be analyzed.

Keywords: *Flowshop Scheduling Problem*, *greedy* heuristics, GRASP metaheuristics, optimization, algorithms, *flowshop*, computational analysis.

Índice general

Resumen **III**

Abstract **V**

Lista de figuras **IX**

Lista de tablas **XI**

1. Introducción **1**

1.1. Contexto 1

1.2. Motivación 2

1.3. Objetivos 2

1.4. Estructura de la memoria 2

2. Flow-shop Scheduling Problem (FSP) **5**

2.1. Scheduling 5

2.2. Flow-shop Scheduling 5

2.3. Permutation Flow-shop Scheduling Problem con necesidad de recursos adicionales 6

2.4. Características del problema 6

3. Revisión bibliográfica **7**

3.1. Heurísticas simples 7

3.2. Heurísticas compuestas 8

3.3. Metaheurísticas 9

4. Formalización del modelo del problema **11**

4.1. Notación 11

4.2. Modelo matemático 12

5. Algoritmo de solución propuesto **15**

5.1. Método constructivo 16

5.2. Método de reparación 19

5.3. Procedimiento de mejora estocástico 22

5.3.1. Búsqueda local 23

5.3.2. Greedy Randomized Adaptive Search Procedure (GRASP) 24

6. Experimentos computacionales **27**

6.1. Experimentos computacionales con el criterio de menor tiempo de ejecución en máquina 1 30

6.2. Experimentos computacionales con la heurística basada en Newaz, Enscore y Ham (NEH) 31

6.3. Análisis comparativo de los métodos 35

6.3.1. Comparación de métodos para el criterio basado en menor tiempo de ejecución en máquina 1 36

6.3.2. Comparación de métodos para el criterio basado en NEH . . 38

7. Conclusiones y tabajo futuro **43**

Bibliografía **46**

Lista de Figuras

5.1. Trabajo 1 asignado en la secuencia. 17

5.2. Trabajo 5 seleccionado. 18

5.3. Solución inicial sin reparar. 18

5.4. Retraso de la Tarea 1. 20

5.5. Retraso innecesario de la Tarea 3. 21

5.6. Solución factible reparada. 21

6.1. Boxplot de los RPD obtenidos para el criterio de menor tiempo de ejecución en máquina 1. 38

6.2. Boxplot de los RPD obtenidos para el criterio NEH. 41

Lista de Tablas

5.1. Tiempos de ejecución (p_{ij}) para dos máquinas y seis trabajos. 17

5.2. Recursos (r_{ij}) para dos máquinas y seis trabajos. 17

6.1. Tiempos de ejecución en segundos obtenidos para el criterio de menor tiempo de ejecución en máquina 1. 30

6.2. *Makespan* obtenidos para el criterio de menor tiempo de ejecución en máquina 1. 30

6.3. Tiempos de ejecución obtenidos para el criterio NEH. 33

6.4. *Makespan* obtenidos para el criterio NEH. 33

6.5. RPD obtenidos para el criterio de menor tiempo de ejecución en la primera máquina. 36

6.6. RPD obtenidos para el criterio NEH. 39

Capítulo 1

Introducción

1.1. Contexto

Actualmente, las empresas productoras se encuentran en constante búsqueda de maneras de maximizar las producciones de sus servicios para obtener un mayor número de beneficios. Es por eso que la investigación operativa juega un importante papel en dichos contextos, ya que se encarga de la optimización de los procesos productivos. Así pues, se quiere obtener la manera de aprovechar de la mejor forma posible los recursos disponibles a la hora de asignarlas a tareas correspondientes en el ámbito empresarial. Un ejemplo puede ser la asignación de tareas a máquinas en fábricas minimizando el coste de tiempo de ejecución y otros factores como el número de operarios disponibles.

A lo largo de este trabajo se introducirá un problema de ajuste de tareas a máquinas a ejecutar en el mínimo tiempo posible a través de diferentes algoritmos implementados y analizaremos los resultados para poder realizar inferencias y sacar conclusiones.

En este caso, consideraremos un número de tareas que deberán ser ejecutadas en un orden predefinido por un número de máquinas en las cuales cada tarea puede tardar un tiempo diferente y consumir una cantidad determinada de recursos. Consideraremos que cada máquina puede ejecutar una tarea a la vez y que una tarea no puede ser ejecutada en más de una máquina a la vez. Además, tendremos en cuenta un número de recursos máximos los cuales no podremos exceder en ningún momento de la producción.

1.2. Motivación

El móvil de este trabajo es poder obtener un método de búsqueda de soluciones factibles para este problema de optimización que podrá ser utilizado en procesos de fabricación de empresas que necesiten optimizar sus recursos temporales y de personal.

1.3. Objetivos

Los objetivos que se proponen lograr con la realización de este trabajo son los siguientes:

- Introducir el problema de *Permutation Flow-shop scheduling (P-FSP)* y presentar un modelo matemático formalizado para este.
- Profundizar en conceptos de la investigación operativa y en la resolución de problemas de optimización aplicados.
- Búsqueda de una solución factible a un problema real a través de diferentes técnicas como algoritmos con diferentes criterios greedy, uso de aleatorización y metaheurísticas GRASP.
- Realización de experimentos computacionales para comparar los resultados obtenidos con diferentes modificaciones del algoritmo y con diferentes instancias del problema.
- Aprender a interpretar los resultados obtenidos, mostrarlos de forma gráfica y tomar decisiones en base a ellos.

1.4. Estructura de la memoria

Este documento se estructura de la siguiente forma:

Introducción: Presentación del problema a resolver y su contexto, así como la motivación y los objetivos del trabajo.

Flowshop Scheduling Problem (FSP): En este capítulo se introducirá el problema de ajuste de tareas a máquinas a ejecutar en el mínimo tiempo posible y se explicará en qué consiste, además de explicar sus características, conceptos y un supuesto práctico que es el que resolveremos más adelante.

Revisión bibliográfica: Estado del arte de la investigación operativa y el ajuste de tareas a máquinas, así como una revisión de los algoritmos utilizados en este trabajo además de otras opciones.

Definición formal del problema: En este capítulo se formaliza el problema a resolver y se presentan los supuestos matemáticos y restricciones que se tendrán en cuenta a la hora de resolver instancias pequeñas.

Algoritmo Propuesto: En este capítulo se presenta el algoritmo propuesto para la resolución del problema, así como su implementación y los criterios utilizados para la búsqueda de soluciones.

Experimentos computacionales: En este capítulo se presentan los experimentos computacionales realizados para la comparación de los resultados obtenidos con diferentes modificaciones del algoritmo y con diferentes instancias del problema. Se analizarán los resultados obtenidos y se mostrarán gráficamente.

Conclusiones: En este capítulo se presentan las conclusiones y líneas de trabajo futuras para llevar a cabo y profundizar en el tema tratado.

Capítulo 2

Flow-shop Scheduling Problem (FSP)

2.1. Scheduling

Los problemas de planificación *(scheduling)* son problemas que surgen en diferentes contextos y que intentan resolver la necesidad de realizar diferentes trabajos o actividades por determinadas entidades capaces de realizarlas con el propósito de optimizar un objetivo. Estos problemas se puede esquematizar de una manera sencilla con tres componentes: ¿qué?, ¿quién? y ¿cuándo?.

En este Trabajo de Fin de Grado nos centraremos en los contextos de la producción y fabricación, es decir, la asignación de tareas a máquinas y hablaremos siempre de estos dos tipos de recursos a lo largo de la siguiente memoria.

2.2. Flow-shop Scheduling

Un *flow-shop* es una disposición de máquinas en un proceso productivo de una fábrica. En ellas contamos con un conjunto de m de máquinas $I = \{1, 2, ..., m\}$ que son dispuestas en serie con el objetivo de lograr un producto final.

El problema de *Flow-shop Scheduling (FSP)* es un problema de optimización en las ciencias de la computación y más en concreto en la rama de investigación operativa. Se trata de un problema de planificación del orden de n trabajos de $J = \{1, 2, \ldots, n\}$ que se tienen que ejecutar como tareas en m máquinas. Cada tarea tiene un tiempo de ejecución en cada máquina y el objetivo es encontrar el orden de procesado de las tareas en las máquinas que minimice el tiempo total

de ejecución. Este tiempo de ejecución total desde el inicio hasta el final de la producción es el que llamaremos a partir de ahora como *makespan* [1].

2.3. Permutation Flow-shop Scheduling Problem con necesidad de recursos adicionales

El problema específico que vamos a tratar a lo largo del trabajo es el *Permutation Flow-shop Scheduling Problem*, es decir, cada máquina es una etapa en el proceso de producción y los productos traspasan de una máquina a otra, hasta recorrer todas las etapas o máquinas en secuencia y en el mismo orden de tareas establecido al principio de la cadena [2].

Contamos con un número n de trabajos de un conjunto $J = \{1, 2, ..., n\}$. Estos trabajos, se ejecutan en máquinas y siguen el mismo orden de paso por las máquinas.

Además, para añadir un punto más de dificultad, vamos a tener en cuenta un tipo más de recursos definido de antemano, de tal manera que en cada instante, no se podrá exceder el número de dichos recursos. Este tipo de recursos pueden ser por ejemplo el número de operarios en las fábricas de tal manera que cada ejecución de una tarea en una máquina requiere de un número de operarios.

2.4. Características del problema

Como hemos dicho anteriormente, contamos con un conjunto de trabajos $J = \{1, 2, ..., n\}$ y un conjunto de máquinas $I = \{1, 2, ..., m\}$ a las cuales se le asignarán las tareas correspondientes en un orden determinado y en unos tiempos determinados.

Cada tarea dentro del conjunto a optimizar requiere un tiempo de ejecución conocido, determinista y no negativo en cada máquina denotado como p_{ij} $j \in J$, $i = 1, 2, ..., m$. También, cada tarea necesita un número de recursos r_{ij} para ser ejecutada en la máquina i y contaremos con un r_{max} que será el límite de recursos del cual no se puede exceder la producción en ningún momento.

Para el problema de flow-shop, suponemos que las tareas son independientes entre sí y que todas están disponibles para procesar en el tiempo 0 sin tener en cuenta ningún tiempo de pre-proceso de tarea ni tiempo de preparación de la máquina. Además, las máquinas están disponibles durante todo el proceso de producción y una tarea no puede ser procesada en más de una máquina a la vez ni una máquina puede procesar más de una tarea en el mismo lapso de tiempo.

Capítulo 3

Revisión bibliográfica

En el siguiente capítulo, vamos a presentar la bibliografía utilizada para la redacción de esta memoria. En este trabajo, como se ha explicado en el Capítulo 2, se pretende proponer una solución computacional al problema de *Flowshop Scheduling Problem* (FSP) con la consideración de recursos adicionales. Si bien es cierto que existe información adherida al problema FSP, no existe información suficiente para explicar el mismo problema con esta consideración de recursos extra. Por este motivo, a continuación realizaremos un recorrido por diferentes soluciones relacionadas con el FSP y que nos han ayudado para la consecución de un correcto desarrollo de este trabajo.

Framinan en [3], divide las diferentes heurísticas en dos grupos: métodos simples y compuestos. Una heurística, normalmente suele estár formada por varias fases para aplicarse, como una fase de construcción de una solución inicial, seguida de una fase de mejora de esta solución inicial. Las heurísticas compuestas son llamadas a los métodos que en sus fases contienen otras heurísticas para resolver subproblemas, mientras que las heurísticas simples, se las llaman a las heurísticas que no contienen más heurísticas en alguna de sus fases.

3.1. Heurísticas simples

Un ejemplo de heurística simple es la que propone Campbell en [3]. La heurística CDS (Campbell Dudek Smith) es una heurística que se basa en el algoritmo de Johnson, siendo extensión de este. El algoritmo de Johnson solo puede ser aplicado a problemas de dos máquinas, que va ordenando la tareas en función de los tiempos de ejecución en la máquina primera y la segunda. En este caso, la heurística CDS, se generan $m-1$ situaciones ficticias utilizando dos máquinas virtuales en cada una de las situaciones, donde m es el número de máquinas. En cada una de estas situaciones ficticias, se aplica el algoritmo de Johnson y se obtiene una secuencia de tareas

ordenadas. La secuencia que minimiza el tiempo total de ejecución es la solución del problema. Aunque este algoritmo tiene una complejidad de $O(m^2)$, no es muy intersante ya que no es capaz de resolver instancias grandes de este problema. Gupta [4] introdujo tres heurísticas simples, *minimo tiempo de espera (MINIT)*, *mínimo tiempo de finalización (MICOT)* y algoritmos MINIMAX, y comparó con los resultados de la heurística CDS obteniendo unos mejores resultados con menor tiempo computacional.

Existen otras heurísticas simples que también ofrecen soluciones a este tipo de problema, como Krone y Steiglitz [5], Miyazaki [6]; y Miyazaki y Nishiyama [6]. Sin embargo, la mejor solución propuesta para el problema PFSP es la heurística NEH (Nawaz, Enscore y Ham) en [7], con el criterio de *makespan*. Se basa en la idea de que los trabajos con mayor tiempo de procesamiento deben ser establecidos lo más pronto posible. Así, la heurística genera un orden inicial de trabajos de manera decreciente, en función de su suma total de tiempos de procesado o *makespan parcial*. Luego, se van creando secuencias de trabajos construídas a partir de la evaluación de todas las secuencias obtenidas de insertar un trabajo desde el inicio hasta todas las posiciones posibles de la secuencia parcial actual. Este algoritmo cuenta con una complejidad de $O(n^3m)$, que según los experimentos realizados hasta el momento, es la heurística más efectiva de todas.

3.2. Heurísticas compuestas

Liu y Reeves en [8] proponen un esquema de mejora basado en intercambios de trabajos. Empezando por una secuencia inicial, el proceso trata de intercambiar cada trabajo con un cierto número de trabajos. Si la mejor secuencia es obtenida a partir de estos intercambios, mejorando la secuencia actual, esta se reemplaza. Después de probar todos los trabajos, el proceso vuelve a iniciarse desde el primer trabajo de la secuencia. Este se repite hasta que no se encuentra mejora en una ronda de ejecución. Este proceso se llama *Forward Pairwise Exchange* (FPE). La versión contaria de este procedimiento es conocida como *Backward Pairwise Exchange* (BPE), que recorre la secuencia en el sentido de derecha a izquierda.

Allahverdi y Aldowaisan en [9] proponen un total de siete heurísticas compuestas resultantes de combinar algoritmos NEH, WY (Woo y Yim [10]) y RZ (Rajendran y Ziegler [11]) con procedimientos de búsqueda local, incluyendo FPE con reinicio y búsqueda local de la heurística RZ. Estas heurísticas se proponen debido a un mejor rendimiento con respecto de otras.

Sin embargo, no entramos mucho en detalle puesto que no es nuestro objetivo principal del problema, nos centraremos más en heurísticas simples y en su funcionamiento, así como heurísticas compuestas mucho más simples.

3.3. Metaheurísticas

Una metaheurística es un procedimiento global, que trata de buscar una solución a un problema a través de múltiples heurísticas y técnicas que se aplican de manera flexible. Su proposito es optimizar problemas complejos, no atrapándose en óptimos locales.

En cuanto a metaheurísticas aplicadas al problema de FSP, podemos mencionar diferentes propuestas utilizando diferentes criterios. Métodos como la búsqueda de *Noteworthy Tabu* de Nowicki y Smutnichi en [12]. Tambíen métodos de *simulated annealing* son propuestos por Osman y Potts en [13], mientras que algoritmos genéticos adaptados al problema también son presentados por Reeves en [14].

En este trabajo, nos centraremos más en una metaheurística concreta, la metaheurística GRASP (*Greedy Randomized Adaptative Search Procedure*), que es una metaheurística que combina una fase de construcción de solución inicial aleatoria y una fase de mejora de esta solución. Esta metaheurística se basa en la idea de que la solución inicial puede ser mejorada a través de una búsqueda local, y que la implementación de la aleatoriedad en la construcción inicial puede ayudar a evitar óptimos locales. Más adelante, entraremos más en detalle en esta metaheurística y su funcionamiento aplicado al problema P-FSP con recursos adicionales.

Capítulo 4

Formalización del modelo del problema

En esta sección vamos a dar una definición más precisa y detallada sobre el problema de *Permutation Flow-shop scheduling (P-FSP) con recursos adicionales* y se propone un modelo matemático para este. Además, se profundizará en conceptos de la investigación operativa y la resolución de problemas de optimización aplicados.

Previamente, vamos a definir los conceptos de tareas sucesoras y predecesoras.

Definición 1. Se dice que una tarea k es sucesora de una tarea j si son ejecutadas en una misma máquina y entre el final de la ejecución de la tarea j y el inicio de la ejecución de la tarea k no se procesa ninguna tarea en la misma máquina. La tarea j es predecesora de la tarea k.

Definición 2. Dos tareas j y k son sucesivas o consecutivas si la tarea j es predecesora de la tarea k o viceversa.

4.1. Notación

A continuación vamos a presentar la notación necesaria para plantear el problema de P-FSP con recursos adicionales.

En primer lugar, necesitamos dos conjuntos de datos para representar el número de máquinas y de tareas con los que vamos a trabajar y asignar, estos conjuntos son:

- $I = \{1, 2, ..., m\}$: conjunto de máquinas idexadas por la letra i.
- $J = \{1, 2, ..., n\}$: conjunto de tareas idexadas por la letra j o k.

Una vez definidos el número de máquinas y de tareas que tenemos, es necesario definir la notación necesaria para representar el tiempo de ejecución de cada tarea en cada máquina y el número de recursos que necesita cada tarea en cada máquina. Esto lo representaremos con dos matrices, una para los tiempos de ejecución y otra para los recursos. Estas matrices son P y R respectivamente con los consiguientes parámetros:

- p_{ij}: tiempo de ejecución de la tarea j en la máquina i.
- r_{ij}: número de recursos adicionales que necesita la tarea j en la máquina i.

También necesitaremos una constante R_{max} que será el número máximo de recursos que se pueden utilizar en cada instante de tiempo.

Cabe recordar, que en este trabajo, vamos a suponer que las tareas no necesitan ningún tiempo de preparación y que las máquinas están disponibles durante todo el proceso de producción.

4.2. Modelo matemático

En esta sección se propone un modelo matemático obtenido de la tesis de Pedro Alfaro Fernandez en [15]. El modelo que se muestra en dicha tesis, llamado PFSPR-Scheduling, es una adaptación de un modelo de máquinas paralelas no relacionadas, propuesto en UPMR-S de Fanjul-Peyro, Perea y Ruiz (2017) [16], ajustado al problema de Flow-shop.

Para formular el problema para el modelo de programación lineal, vamos a considerar los siguientes parámetros y variables de decisión además de los explicados anteriormente:

- Parámetro K_{max}: marca el horizonte temporal en el modelo para el índice k, es una cota superior. Más adelante se detalla el cálculo de K_{max}.
- Variables binarias x_{ijk}: toma el valor 1 si la tarea del trabajo j procesada en la máquina i acaba su procesamiento en el instante de tiempo k. En caso contario $x_{ijk} = 0$
- Variable continua C_{max}: tiempo de finalización en la última máquina m del último trabajo.
- Variable binaria F_{jg}: variable con valor 1 cuando el trabajo j se procesa después que el trabajo g, y 0 en caso contrario. Al ser un problema con permutación esto implica que cada una de las tareas i del trabajo j tendrá que procesarse antes que cada una de las tareas i del trabajo g.

El modelo propuesto es como sigue:

$$\text{Minimize} \quad C_{max} \tag{4.1}$$

$$\sum_{k \geq p_{mj}} k x_{mjk} \leq C_{max} \quad \forall j \in J \tag{4.2}$$

$$\sum_{k \geq p_{ij}} x_{ijk} = 1 \quad \forall i \in I, \forall j \in J \tag{4.3}$$

$$\sum_{k \geq p_{ij}} x_{ijk} \leq \sum_{l \geq p_{ij}} x_{i+1,j,l-p_{i+1,j}} \quad \forall j, i \in 1 \dots m-1 \tag{4.4}$$

$$\sum_{j} \sum_{s \in \text{máx}(k,p_{i,j}),\dots,k+p_{i,j}-1} x_{i,j,s} \leq 1, \quad \forall i,k \tag{4.5}$$

$$\sum_{i} \sum_{j} \sum_{s \in \text{máx}(k,p_{i,j}),\dots,k+p_{i,j}-1} r_{i,j} x_{i,j,s} \leq R_{max}, \quad \forall k \tag{4.6}$$

$$\sum_{i} \sum_{j} x_{ijk} \leq m, \quad \forall k \tag{4.7}$$

$$\sum_{k \geq p_{ij}} k x_{ijk} \leq \sum_{l \geq p_{ig}} l x_{igl} + M F_{jg}, \quad \forall i,j,g, g \neq j \tag{4.8}$$

$$F_{jg} + F_{gj} = 1, \quad \forall j,g, g \neq j \tag{4.9}$$

El objetivo principal es la minimización del tiempo de finalización máximo, conocido como *makespan* definido por la ecuación 4.1. La ecuación 4.2 asegura que ningún trabajo pueda terminar después de C_{max}, ajustando el tiempo de finalización de la última tarea del último trabajo en la última máquina. La restricción 4.3 fuerza que todos los trabajos se ejecutan una sola vez en cada una de las máquinas. La ecuación 4.4 obliga a cumplir la precedencia de las tareas en un flujo, es decir, para un trabajo j, su tarea en la máquina i tiene que estar finalizada antes que su tarea en la máquina $i+1$. La restricción 4.5 asegura que ninguna máquina procesa más de un trabajo en un instante de tiempo. La ecuación 4.6 obliga a que la suma de los recursos utilizados en cualquier instante de tiempo no sobrepase la capacidad de recursos máximos en el sitema R_{max}. La restricción 4.7 fuerza que no haya más máquinas activas de las m máquinas que hay en el taller. La restricción 4.8 valida el orden de los trabajos, haciendo que la variable F_{jg} sea 0 siempre que el trabajo j preceda al trabajo g en todas las máquinas y 1 en caso contrario. La restricción 4.9 asegura que solo uno de los trabajos precede al otro.

K_{max} se define como un valor entero positivo, una cota superior (*Upper Bound*) que delimita el horizonte temporal del problema en cuestión.

En un caso con permutación y sin recursos (PFSP), se propone la cota superior de la expresión 4.10, que consiste en encontrar el tiempo de procesado acumulado en la máquina que toma más tiempo total en procesar todos los trabajos máx(p_i), obtener el tiempo de procesado total que necesita el trabajo más costoso máx(p_j) y juntarlo en un *schedule* además de restarle el tiempo de proceso de la tarea i en el trabajo j, porque esta tarea se ha sumado dos veces, tanto en la máquina como en el trabajo.

$$\text{máx}(p_i) + \text{máx}(p_j) - p_{ij} \tag{4.10}$$

Al contar con recursos adicionales esta cota no es válida, a no ser que haya más recursos disponibles de los que se necesitan.

$$\text{máx}(r_{ij}) \cdot h \leq R_{max} \tag{4.11}$$

Si para el número de máquinas disponibles se cumple la condición 4.11, que todas las máquinas trabajando simultáneamente en tareas que tienen un requisito de recursos que es el máximo del problema no superan los recursos disponibles (R_{max}), entonces los recursos no influyen en el problema y se puede utilizar la cota que se plantea para (PFSP).

Este modelo matemático, como otros con la misma finalidad para este problema o parecidos, suelen tener muchas limitaciones en la capacidad de resolución, resolviendo la optimalidad de un problema de reducido tamaño pero siendo incapaz de resolver problemas de tamaño medio o grande. Por ello es necesario utilizar otros métodos de resolución y que pasaremos a presentar a continuación.

Capítulo 5

Algoritmo de solución propuesto

En la presente sección, vamos a explicar un algoritmo heurístico para el problema de P-FSP con recursos adicionales. Este algoritmo que se explica a continuación propone una solución que se compone de dos fases o partes diferenciables.

La primera fase de construcción de una solución inicial que cumple con las restricciones en cuanto orden y uso de máquinas pero en la que no se tendrán en cuenta los recursos utilizados en cada instante de tiempo. Y una segunda fase en la que se reparará la solución obtenida en la primera fase para que cumpla con las restricciones de recursos máximos por unidad de tiempo impuestas en el problema.

Se ha elegido aplicar este algoritmo debido a la gran complejidad del problema y la imposibilidad de obtener una solución óptima en un tiempo razonable. La combinación de estas fases nos permitirá obtener soluciones que combinan rapidez y calidad dentro de lo posible. La construcción de la solución inicial permite explorar el marco de soluciones posibles, dando un punto del que partir. Más adelante explicaremos una fase de mejora de esta solución inicial que nos permitirá explotar dicha solución inicial. Por último, la fase de reparación nos permitirá asegurar una solución factible y por tanto aceptable y comparable. Estas fases nos dotan de una mayor robustez al algoritmo, reduciendo el riesgo de obtener soluciones subóptimas o inválidas.

En las siguiente secciones 5.1 y 5.2 se explicarán en detalle cada una de las fases del algoritmo propuesto y además se propondrá una aleatorización del algoritmo para obtener soluciones diferentes en cada ejecución del algoritmo buscando la mejor solución posible. Ver sección 5.3.

5.1. Método constructivo

El método constructivo propuesto para la resolución del problema de P-FSP con recursos adicionales se basa en la utilización de un algoritmo *greedy* que construye una solución inicial factible para el problema. Podremos determinar el mejor criterio *greedy* para el algoritmo pero en esta sección el criterio que utilizaremos se basa en la elección de la tarea que menos tiempo de ejecución tenga en la primera máquina de la secuencia.

Este criterio es elegido en esta sección con el objetivo de simplificar los cálculos y poder dar una explicación más sencilla sobre el problema a tratar. Más adelante también trataremos heurísticas como la NEH (Nawaz, Enscore y Ham) modificada, que se basa en la elección de la tarea que minimiza el makespan en cada iteración.

Esta heurística *greedy* se basa en el proceso de selección del mejor trabajo posible que no haya sido asignado y calcular el tiempo de finalización de cada tarea en cada máquina en secuencia para poder determinar el makespan al insertar dicho trabajo.

A continuación, en el Algoritmo 1 se presenta el pseudocódigo del algoritmo *greedy* para la construcción de la solución inicial.

Algoritmo 1: Greedy fase constructiva.

```
for t ∈ tareas do
    for i ∈ tareas_no_asignadas do
        if p_i1 < p_min then
            tiempo_minimo ← p_i1;
            tarea_minima ← i;
    orden(t) ← tarea_minima;
    Eliminar de tareas_no_asignadas
    for j ∈ maquinas do
        Establecer tiempo de inicio y fin
    for t ∈ [tiempo_inicio, tiempo_fin] do
        Establecer uso de recursos en el instante t
```

Se presenta el siguiente supuesto práctico para ofrecer claridad sobre el problema que vamos a tratar.

Ejemplo 1. Supongamos que tenemos un conjunto de trabajos $J = \{1, 2, 3, 4, 5, 6\}$, un conjunto de máquinas $I = \{1, 2\}$ y una restricción de recuros máximos en cada instante de tiempo $R_{max} = 3$. Nos proporcionan los tiempos de ejecución de las tareas de cada trabajo en cada máquina y el número de recursos que necesita cada tarea en cada máquina. Esta información se presenta en las Tablas 5.1 y 5.2.

	J1	J2	J3	J4	J5	J6
M1	12	24	22	19	18	19
M2	19	19	17	21	22	8

Tabla 5.1: Tiempos de ejecución (p_{ij}) para dos máquinas y seis trabajos.

	J1	J2	J3	J4	J5	J6
M1	3	1	2	1	2	1
M2	2	2	2	2	1	2

Tabla 5.2: Recursos (r_{ij}) para dos máquinas y seis trabajos.

Comenzamos el algoritmo de manera que se recorren todas los trabajos de N (*tareas_no_asignadas* en el pseudocódigo), que vamos a suponer que es la lista de trabajos que todavía no han sido asignados, para seleccionar el que menor tiempo de ejecución tenga en la primera máquina. En este caso, el trabajo 1 es el que menos tiempo de ejecución tiene su correspondiente tarea en la primera máquina, por lo que se selecciona este trabajo para ser el primero en la secuencia, el cual se elimina de la lista de posibles y se añade al orden, quedando como en la Figura 5.1.

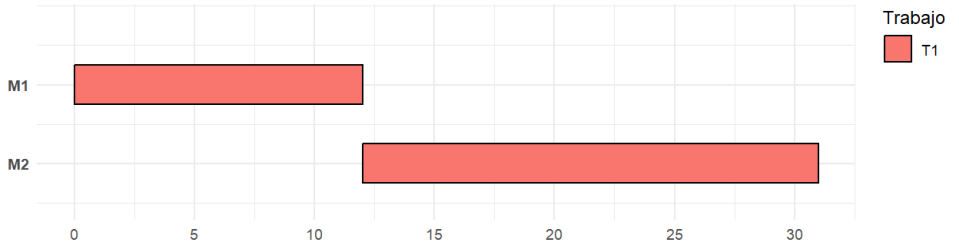

Figura 5.1: Trabajo 1 asignado en la secuencia.

Una vez asignado, tenemos que cuadrar los tiempos de ejecución de cada tarea del trabajo en cada máquina. Para ello, se recorren todas las máquinas y se calcula el instante en el que termina dicha tarea en dicha máquina. En este caso, el trabajo 1 termina en la máquina 1 en el instante 12 y en la máquina 2 en el instante 31.

Además registramos en un array temporal, que llamaremos *recursos_en_tiempo*, la cantidad de recursos que se están utilizando en cada instante de tiempo en cada máquina, en este caso desde el instante 1 hasta el 31. Como en un primer momento no reparamos la solución, simplemente los almacenamos en un array, el cual utilizaremos en la fase de reparación.

Así, ya tendríamos el primer trabajo asignado y cuadrado en el proceso de producción. En la siguiente iteración, repetimos el proceso y extraemos de N el trabajo que menor tiempo de ejecución necesite en la primera máquina, en este

caso el trabajo 5. Una vez encontrado, lo registramos en la secuencia y calculamos los tiempos de ejecución en cada máquina como hemos mencionado anteriormente, resultando como en la Figura 5.2.

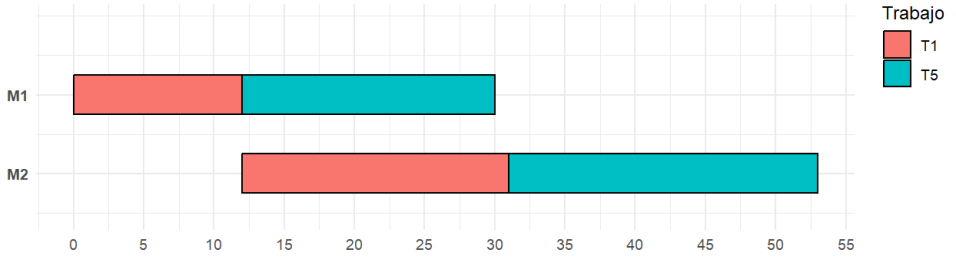

Figura 5.2: Trabajo 5 seleccionado.

El proceso se repite de esta manera hasta que no queden trabajos en el conjunto N, en cuyo caso, ya tendríamos una solución inicial sin tener en cuenta los recursos y que quedaría como en la siguiente Figura 5.3.

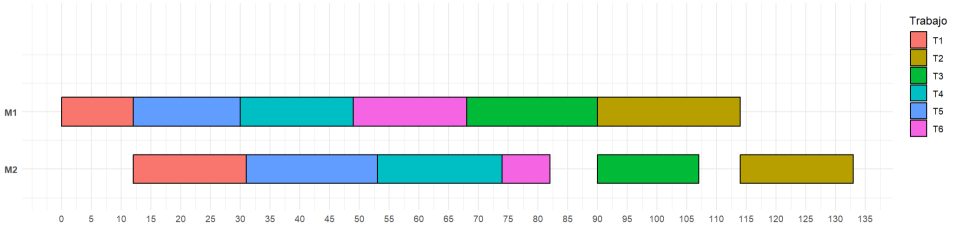

Figura 5.3: Solución inicial sin reparar.

Como podemos ver en la Figura 5.3, obtenemos una solución inicial de 133 unidades de tiempo de makespan. Observamos como existen tiempos en los que la máquina 2 no puede ejecutar ninguna tarea, ya que una de las propiedades del Permutation Flow-shop Scheduling Problem, es que una tarea no puede ser ejecutada en la máquina $i+1$ hasta que termine la ejecución en la máquina i. Además, esta solución no cumple con la restricción de recursos máximos en cada instante de tiempo, asi que tendremos que repararla, procedimiento que vamos a tratar en profundidad a continuación.

5.2. Método de reparación

Lo siguiente que debemos hacer para obtener una solución factible es reparar la solución inicial obtenida en la fase de construcción, de tal manera que cumpla con las restricciones de recursos máximos en cada instante de tiempo. Una de las características más importantes de este problema es que en toda la línea de ejecución de los trabajos en las máquinas, no se pueden exceder unos recursos máximos impuestos de antemano, es decir, en cada instante de tiempo no se pueden utilizar más recursos de los que disponemos, que bien podría asimilarse a la cantidad de operarios disponibles en un entorno industrial. Es así que debemos reparar esta solución inicial que hemos obtenido en la etapa anterior y que no cumple con esta restricción.

Para ello, vamos a utilizar un algoritmo de reparación que se basa en la detección del primer instante de tiempo en el que se incumple la restricción, analizando que tareas están en conflicto y retrasando el tiempo de inicio en una unidad de la tarea cuya ejecución tenga lugar en una máquina más avanzada en la secuencia, es decir, la mayor máquina en cuanto a orden. Por lo que iremos retrasando las tareas en conflicto en una sola unidad en cada iteración hasta que se cumpla la restriccón R_{max}.

Antes de comenzar con la explicación del algoritmo tenemos que tener en cuenta en la fase de construcción, el número de recursos que se está haciendo uso en cada instante. Para ello, recordamos que después de cada inserción en el orden de producción y cálculos de tiempos de ejecución, se va almacenando en un array los recursos que se están utilizando en cada instante de tiempo entre el inicio hasta el final de cada tarea en cada máquina, añadiéndolos a los que ya haya. Este array lo llamaremos *recursos_en_tiempo()* y lo utilizaremos en la fase de reparación para detectar los instantes de tiempo en los que se incumple la restricción de recursos máximos.

El Algoritmo 2 presenta el pseudocódigo del algoritmo de reparación:

Algoritmo 2: Fase de reparación.

```
while t < C_max do
    if recursos_en_tiempo(t) > R_max then
        tareas_en_conflicto ← tareas_en_conflicto_en(t);
        tarea_a_retrasar ← tarea más avanzada en conflicto;
        Comprobar si hay tareas contiguas que empujar;
        Retrasar una unidad de tiempo tarea_a_retrasar y sus contiguas;
        Actualizar recursos_en_tiempo() ;
        Actualizar C_max;
    t++;
```

Es importante tener en cuenta que cuando retrasamos una tarea, nuestro array de *recursos_en_tiempo()* se ve afectado, ya que al retrasar una tarea, estamos

liberando recursos en un instante de tiempo en el que antes no lo hacíamos. Por lo que tenemos que volver a calcular los recursos utilizados en cada instante de tiempo para poder detectar el siguiente instante de tiempo en el que se incumple la restricción de recursos máximos. Como podemos ver en el código.

Ejemplo 2. Siguiendo con el ejemplo anterior, vamos a reparar la solución obtenida en la fase de construcción. Para ello, vamos a analizar el array *recursos_en_tiempo()* que hemos obtenido en la fase de construcción y vamos a detectar el primer instante de tiempo en el que se incumple la restricción de recursos máximos.

En este caso, el primer instante de tiempo que detectamos que se incumple con la restricción de recursos es el instante 12, en el que se están utilizando 4 recursos en total entre la *Tarea 5* en la máquina 1 y la *Tarea 1* en la máquina 2.

Así pues, como hemos explicado anteriormente, una vez obtenido el conjunto de tareas en conflicto, retrasamos el tiempo de inicio de la tarea que se encuentra en la máquina más avanzada en la secuencia, en este caso la *Tarea 1* en la máquina 2, en una unidad de tiempo. Cómo la *Tarea 5* ocupa el intervalo temporal desde el 12 hasta el 30 en la máquina 1, tendremos que retrasar la *Tarea 1* en la máquina 2 hasta el instante de tiempo 30. Como podemos ver en la Figura 5.4.

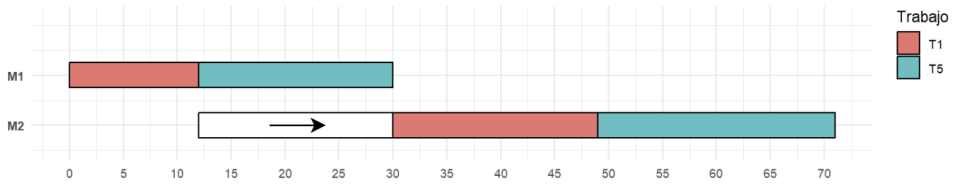

Figura 5.4: Retraso de la Tarea 1.

Una vez atrasada la tarea se vuelve a calcular los recursos en cada instante de tiempo con la nueva solución provisional y se procede a detectar cuál es el siguiente instante de tiempo en el que se incumple con la restricción.

Cabe destacar un caso muy importante en el algoritmo de reparación, y es que si en la fase de construcción se ha tenido que retrasar el inicio de una tarea debido a que la tarea no había terminado de ejecutarse en la máquina anterior, en la fase de reparación puede que esto se haya solucionado por sí solo y que ahora tengamos espacios en la secuencia de ejecución en los que ninguna tarea está ejecutando. En nuestro ejemplo, esta situación ocurre, de manera que la *Tarea 3*, que en la fase constructiva estaba retrasada por su ejecución en la máquina anterior, ahora este problema se ha solucionado. Si no tuvieramos en cuenta este caso, las tareas que fueron retrasadas por este motivo antes de la reparación empezarían más tarde de lo que en realidad podrían empezar. Este caso queda reflejado en la Figura 5.5,

donde la tarea sigue retrasada en la secuencia, resultando un hueco en el que se podría estar ejecutando dicha tarea y minimizar más el *makespan*.

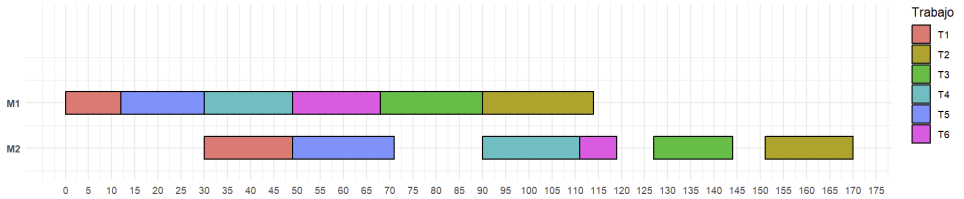

Figura 5.5: Retraso innecesario de la Tarea 3.

Por eso, implementamos en el algoritmo de reparación este caso especial comprobando antes de retrasar una tarea, si la siguiente tarea empieza en el mismo momento en el que la tarea actual termina o si existe un hueco. De esta manera, el algoritmo comprueba esta situación, en el caso de que se de, la tarea se retrasa rellenando ese hueco sin empujar las siguientes tareas en esa máquina. Del modo contrario, si no se da esta situación, se retrasa en una unidad todo el bloque de tareas consecutivas entre sí en la máquina correspondiente.

Si aplicamos este caso especial, vemos como se repara del todo de manera correcta la solución, resultando una solución factible que cumple con las restricciones de recursos máximos en cada instante de tiempo.

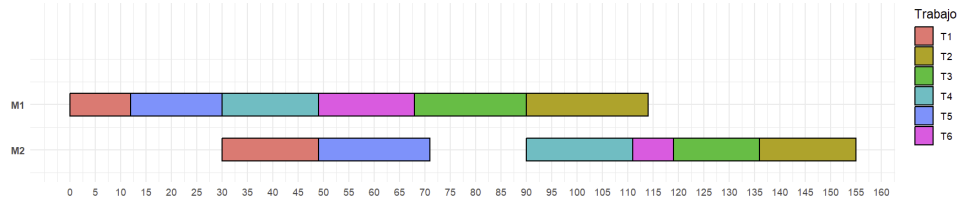

Figura 5.6: Solución factible reparada.

En la Figura 5.6 podemos ver el orden final de las tareas que conjuntan una solución factible para el problema de P-FSP con recursos adicionales. En este caso, el makespan de la solución es de 155 unidades de tiempo, incrementando el obtenido en la fase de construcción, pero ahora cumpliendo con las restricciones de recursos máximos en cada instante de tiempo.

5.3. Procedimiento de mejora estocástico

Con la finalidad de buscar la mejor solución óptima posible, vamos a proponer una aleatorización del algoritmo presentado que nos ayudará en la búsqueda de mejores soluciones. Esta aleatorización se basa en la modificación de la fase de construcción del algoritmo, en la que en lugar de seleccionar simplemente la tarea que menos tiempo de ejecución tenga en la primera máquina, se construirá una Lista de Candidatos Restringida (RCL) de tamaño k. De esta manera, añadiremos a la RCL las k tareas con menor tiempo de ejecución en la máquina 1 y seleccionaremos de manera aleatoria una de las tareas de la RCL para ser añadida al orden de producción. Cuando encontremos una tarea para insertar en la solución, esta la marcaremos como asignada de tal manera que no pueda volver a ser seleccionada para su inserción en la RCL.

De esta manera, conseguimos obtener soluciones diferentes en cada ejecución del algoritmo, ya que la selección de la tarea a insertar en la solución se realiza de manera aleatoria. Incluso podríamos obviar la RCL, de tal manera que la selección de la tarea a insertar en la solución se realice de manera completamente aleatoria de todo el conjunto de tareas no asignadas pero la solución se compondría de una aleatoridad completa. En en el ejemplo que estamos tratando, utilizaremos una RCL de tamaño 3, es decir, la mitad de tareas que tenemos en el conjunto inicial.

El Algoritmo 3 presenta el pseudocódigo de la fase de construcción del algoritmo con aleatorización:

Algoritmo 3: Fase constructiva con aleatorización.

```
for t ∈ tareas do
    Vaciamos la lista RCL;
    Añadimos a tareas_no_asignadas las tareas que no están en la
     solución;
    for h ∈ 1 : k do
        for i ∈ tareas_no_asignadas and i ∉ RCL do
            Obtener la tarea con menor p_i1;
        Añadir a la RCL la tarea con menor p_i1;
    Añadir a la solución una tarea aleatoria de la RCL;
    orden(t) ← tarea_aleatoria;
    Eliminar de tareas_no_asignadas;
    Calcular tiempos de inicio y fin;
```

5.3.1. Búsqueda local

Antes que nada, vamos a introducir una fase más al algoritmo propuesto inicialmente y que consideraremos en la aleatorización. Esta fase se basa en una búsqueda local después de obtener una primera solución no factible, es decir, después de la fase de construcción y antes de la fase de reparación.

A continuación, el Algoritmo 4 presenta en forma de pseudocódigo el algoritmo de búsqueda local que se va a presentar:

Algoritmo 4: Búsqueda local.

```
mejora ← 1;
while mejora ← 1 do
    mejora ← 0;
    for i, j ∈ tareas do
        Intercambiar tareas i y j;
        nuevo_cmax ← calcular C_max;
        if nuevo_cmax < C_max then
            C_max ← nuevo_cmax;
            mejora ← 1;
            Actualizar orden de tareas;
        else
            Intercambiar tareas i y j;
```

Esta búsqueda local lo que hará es intercambiar dos tareas de la secuencia de producción y comprobar si con esta nueva solución obtenemos una mejora en el *makespan*. Si es así, se quedará con la nueva solución y volverá a realizar el proceso de intercambio de tareas con esa nueva solución. Este proceso se repetirá hasta que no se obtenga una mejora en el makespan. En cuyo caso se devolverá la última solución obtenida y que pasará a ser reparada.

Consideramos reparar la solución como último paso del algoritmo. Esto deja como resultado finalmente los siguientes pasos: construcción, búsqueda local y reparación. De esta manera, nuestro primer objetivo es encontrar la mejor solución posible sin tener en cuenta los recursos y más adelante repararla para que cumpla con las restricciones de recursos impuestas en el problema. Así, logramos optimizar en gran parte el algoritmo, al contrario que si reparásemos la solución al final de la fase constructiva y de la búsqueda local. Además de que la pérdida no es significativa, ya que este algoritmo será implementado en la metaheurística GRASP, puediendo ejecutar en muchas iteraciones, aumentando la probabilidad de la mejor solución posible.

5.3.2. Greedy Randomized Adaptive Search Procedure (GRASP)

GRASP (Greedy Randomized Adaptive Search Procedure) es una metaheurística para optimización combinatoria. GRASP es implementado como un proceso iterativo, donde cada iteración consiste en una fase constructiva, donde una solución *greedy* aleatoria es construida, seguida de una fase de búsqueda local que empieza con esa solución inicial y aplica una mejora iterativa hasta que se encuentra una solución óptima local [17].

Consiste en generar soluciones iniciales, por iteraciones que ayudarán a encontrar la solución óptima global. Sin embargo, utilizar soluciones greedy como puntos de inicio para la búsqueda local en un proceso iterativo puede llevar a soluciones locales que no son óptimas. Esto es porque la cantidad de variabilidad que cuentan las soluciones greedy es pequeña y es poco probable que la solución inicial greedy vaya en la dirección correcta para encontrar un óptimo global. Si utilizáramos una regla determinista y no tuvieramos en cuenta la aleatorización, no habría variabilidad en las soluciones iniciales dando lugar a la producción de las mismas soluciones en cada iteración.

Restricted Candidate List (RCL)

La mayoría de impementaciones de GRASP utilizan algún tipo de estructura de construcción RCL basada en valores. En esta estructura, el parámetro RCL α determina el grado de aleatorización en la construcción. Existen estudios donde se concluye que utilizar un parámeto α fijo puede no converger (asintóticamente) a un mínimo global porque el mecanismo de construcción puede excluir las soluciones que son necesarias para encontrar el óptimo global. Existen diferentes remedios para solucionar esto, como un α generado aleatoriamente en cada iteración, o mecanismo de ajuste del parámetro α después de cada iteración. Para simplificar nuestro supuesto práctico, no haremos uso del parámetro α y simplemente nos centraremos en la construcción de la RCL a través del número de trabajos que queremos optimizar [17].

En cuanto al tamaño de la lista de candidatos restringida (RCL), se propone un tamaño dinámico en 5.1 que se adapta la cantidad de trabajos que quedan por asignar en el conjunto completo.

$$K = round(\frac{n_h}{4}) \tag{5.1}$$

Donde n_h es el número de trabajos que quedan por asignar en la lista de trabajos no asignados. De esta manera, si tenemos un número de trabajos muy grande, la RCL será más grande y si tenemos un número de trabajos pequeño, la RCL será más pequeña. Esto nos permitirá obtener una mayor variabilidad en las soluciones

iniciales aunque no tan grande y por lo tanto una mayor probabilidad de encontrar la mejor solución posible.

A continuación, en el Algoritmo 5 se presenta el pseudocódigo del algoritmo GRASP para la resolución del problema de P-FSP con recursos adicionales:

Algoritmo 5: GRASP.

```
for iter ∈ 1 : iteraciones do
    solucion_actual ← Fase constructiva aleatorizada;
    solucion_actual ← Búsqueda local;
    solucion_actual ← Fase de reparación;
    if solucion_actual < C_best then
        C_best ← solucion_actual;
        orden_best ← orden_actual;
Reparar la solución mejorada;
Devolver la solución mejorada;
```

Capítulo 6

Experimentos computacionales

En el presente capítulo vamos a realizar diferentes análisis computacionales para probar las diferentes heurísticas, así como la metaheurística GRASP. En estos análisis se obtendrán los tiempos de ejecución y *makespan* con diferentes instancias del problema, poniendo a prueba su rendimiento y calidad de los resultados obtenidos.

Vamos a dividir el capítulo en dos partes, dedicadas a los criterios *greedy* correspondientes seleccionados que comentamos en capítulos anteriores. Estos son el criterio de *mínimo tiempo de ejecución en máquina 1* y un criterio basado en el que viene dado en la heurística *Nawaz, Enscore y Ham* (NEH) que consiste escoger la tarea que obtiene el menor *makespan* total al añadirla a la secuencia.

El objetivo de estos análisis es comparar los resultados de ejecución de los algoritmos propuestos y obtener conclusiones sobre su rendimiento y calidad, consiguiendo un mejor algoritmo para distintas situaciones que se puedan dar. Se da por supuesto que no se van a obtener resultados óptimos, ya que el problema es NP-Duro y no se puede garantizar la obtención de una solución óptima en un tiempo razonable pero dependiendo de la situación en la que nos encontremos en nuestro problema podemos seleccionar un algoritmo y configuración diferente para obtener un resultado que se acerque lo máximo posible a la solución óptima.

Los experimentos computacionales han sido ejecutados en una máquina *Huawei Matebook 14s* con las siguientes características:

- Procesador: 11th Gen Intel(R) Core(TM) i7-11370H @ 3.30GHz (3.30 GHz)
- Memoria RAM: 16 GB

- Sistema operativo: Windows 11 Home versión 23H2
- Entorno: Xpress Mosel Workbench

Por último, es necesario explicar los algoritmos que vamos a ejecutar para el análisis, con su respectivo criterio de selección y que se corresponde con cada una de las columnas de las tablas de resultados que se presentan más adelante:

- **Greedy:** Algoritmo *greedy* básico que selecciona la tarea en función del criterio utilizado y la añade a la secuencia.
- **Greedy Aleatorizado 500:** Algoritmo *greedy* que selecciona la tarea en función del criterio greedy utilizado, añadiéndola a la **RCL** (Lista de Candidatos Restringida), para más adelante seleccionar aleatoriamente una de las tareas de la lista, añadiendo esta a la secuencia. Al depender de la aleatorización, vamos a iterar dicho algoritmo para obtener la mejor solución posible, en este caso con 500 iteraciones bastará.
- **Greedy + BL:** Algoritmo *greedy* básico que selecciona la tarea en función del criterio y la añade a la secuencia, pero que además aplica una búsqueda local de intercambio de tareas con el objetivo de mejorar el resultado obtenido.
- **GRASP 500:** Algoritmo de tres fases que consiste en una fase de construcción de una solución inicial, una fase de búsqueda local y una fase de reparación de dicha solución. En este caso se ha configurado para realizar 500 iteraciones, aplicando en cada una de ellas la correspondiente búsqueda local de intercambio de tareas y la reparación correspondiente de la secuencia obtenida.

Cabe destacar que en todos los algoritmos está presente la fase de reparación anteriormente comentada, ya que dicha fase es la que nos permite obtener una secuencia válida y factible para el problema que estamos tratando. Por supuesto, se propone una solución de cada algoritmo para un mismo conjunto de datos, pudiendo comparar las diferentes soluciones que obtengamos de los algoritmos.

Nombraremos los archivos de instancias como NxM_inst_X, donde N es el número de tareas, M es el número de máquinas y X es el número de la instancia. Por ejemplo, *6x2_inst_1* se correspondería con una primera instancia de 6 tareas y 2 máquinas. A continuación podemos observar la estructura de las instancias que se han utilizado para realizar los experimentos computacionales.

```
6  2  3

0  12  1  19
0  24  1  19
0  22  1  17
0  19  1  21
0  18  1  22
0  19  1  8

0  3  1  2
0  1  1  2
0  2  1  2
0  1  1  2
0  2  1  1
0  1  1  2
```

Contamos con una primera línea en la que encontramos el número de trabajos, el número de máquinas y el número de recursos máximos disponibles; seguido de enteros que indican las unidades de tiempo que necesitan las tareas para ejecutarse en cada máquina y por último los recursos que necesita cada tarea en cada máquinas para ser ejecutada. Las tablas de tiempos y recursos se componen de filas que representan las correspondientes tareas que se incluyen, con pares (clave, valor) que representan el número de la máquina correspondiente con su tiempo de ejecución o número de recursos necesarios.

Vamos a estructurar los análisis de manera que ejecutaremos las diferentes modificaciones del algoritmo que hemos explicado anteriormente, con diversas instancias del problema, obteniendo una tabla de resultados. Finalmente pasaremos a explicar dichos resultados obtenidos, situaciones en las que se ha comportado mejor o peor y qué conclusiones podemos presentar, pudiendo obtener una mejor solución dependiendo de la situación en la que nos encontremos.

Así pues, pasamos a realizar los experimentos computacionales para analizar el rendimiento de los algoritmos propuestos, en primer lugar con el criterio de selección de menor tiempo de ejecución en máquina 1 y porteriormente con la heurística basada en NEH.

6.1. Experimentos computacionales con el criterio de menor tiempo de ejecución en máquina 1

A continuación se presentan los resultados obtenidos para cada uno de los algoritmos propuestos, así como el tiempo de ejecución del algoritmo y el *makespan* obtenido para el criterio de elección de tareas de menor tiempo de ejecución en la primera máquina. Ver Tabla 6.1 y 6.2.

Recordamos que el *makespan* es el tiempo total solución de ejecución de todas las tareas en todas las máquinas, es decir, el tiempo que tarda la última tarea en ser ejecutada en la última máquina.

Instancia	Greedy	Greedy Aleatorizado 500	Greedy + BL	GRASP 500
6x2_inst_1	0.011	6.518	0.022	12.695
6x2_inst_2	0.018	13.35	0.032	21.645
8x3_inst_1	0.04	26.775	0.075	36.587
8x3_inst_2	0.031	20.416	0.058	28.489
10x4_inst_1	0.069	37.605	0.121	64.03
10x4_inst_2	0.07	34.263	0.136	60.002
20x4_inst_1	0.17	86.206	0.396	232.828
30x4_inst_1	0.275	157.185	0.914	497.338
50x4_inst_1	0.616	342.811	3.759	1494.82
100x3_inst_2	2.418	1196.44	28.494	14427.196

Tabla 6.1: Tiempos de ejecución en segundos obtenidos para el criterio de menor tiempo de ejecución en máquina 1.

Instancia	Greedy	Greedy Aleatorizado 500	Greedy + BL	GRASP 500
6x2_inst_1	155	136	158	136
6x2_inst_2	211	210	211	246
8x3_inst_1	495	488	508	488
8x3_inst_2	318	276	295	275
10x4_inst_1	532	500	579	518
10x4_inst_2	593	554	593	554
20x4_inst_1	1160	1098	1144	1088
30x4_inst_1	1612	1521	1628	1551
50x4_inst_1	2515	2421	2440	2411
100x3_inst_1	5488	5396	5358	5281

Tabla 6.2: *Makespan* obtenidos para el criterio de menor tiempo de ejecución en máquina 1.

Como podemos ver en las tablas obtenidas, el algoritmo que mejores resultados nos arroja, por lo general, ha sido el algoritmo *GRASP* con 500 iteraciones, obteniendo un *makespan* más bajo en la mayoría de los casos y un tiempo de ejecución razonable. Por supuesto, el tiempo de ejecución va a ser bastante mayor en este que en todos los demás, puesto que estamos ejecutando 500 iteraciones del algoritmo. Aún así parece ser la mejor opción, puesto que es el método que más soluciones está comprobando, aumentando la probabilidad de llegar a la óptima gracias a la aleatorización.

No obstante si observamos las primeras instancias, al ser más pequeñas, podemos obtener mejores resultados con solo una ejecución del algoritmo sin necesidad de aleatorizar como es el caso de las dos primeras filas, los algoritmos *Greedy* y *Greedy + BL*. Estas dos columnas siempre obtienen la misma solución al ejecutar al no contar con aleatorización y que como hemos dicho, al ser intancias pequeñas, es posible obtener una solución aceptable pasando por todas las situaciones posibles.

De manera contraria, las instancias más grandes obtenemos siempre una mejor solución con el algoritmo *GRASP* que con los demás algoritmos, ya que al ser instancias más grandes, el número de combinaciones posibles es mucho mayor y por tanto, la aleatorización nos ayuda a obtener una mejor solución.

Por otro lado encontramos el algoritmo *Greedy Aleatorizado*, que obtiene los mejores resultados con instancias pequeñas, ya que por esto, la aleatorización permite recorrer todas las posibilidades de ordenación de los trabajos, dando con la óptima. Esto, por el contrario, no ocurre con las instancias más grandes, ya que en estos casos, el algoritmo obtiene soluciones completamente aleatorias, dificultando la obtención de una buena solución.

6.2. Experimentos computacionales con la heurística basada en Newaz, Enscore y Ham (NEH)

A continuación, vamos a realizar el mismo análisis que venimos presentando anteriormente pero en este caso con una heurística basada en el algoritmo NEH. Esta heurística consiste en seleccionar la tarea que al añadirla a la secuencia, minimiza el *makespan* parcial de la iteración en la que estamos.

La heurística *Newaz, Enscore y Ham* (NEH) se basa en primer lugar, en ordenar las tareas de mayor a menor tiempo total de ejecución o *makespan* (en su paso por todas las máquinas), añadiendo las dos primeras tareas de la lista ordenada y luego ir añadiendo las demás a la secuencia, probando en todas las posiciones disponibles, de forma que se vaya minimizando el *makespan* parcial teniendo en cuenta las tareas que ya se han añadido a la secuencia. En este trabajo, realizaremos una modificación de este algoritmo, de tal manera que iremos seleccionando la tarea que al añadirla a la secuencia en último lugar, vaya minimizando el *makespan* parcial.

Para implementarlo, tenemos que modificar el algoritmo greedy inicial de tal manera que introducimos este nuevo criterio greedy, obteniendo el algoritmo resultante que podemos observar en el Algoritmo 6.

Algoritmo 6: Greedy con criterio basado en NEH.

```
for t ∈ tareas do
    for i ∈ tareas_no_asignadas do
        orden(t) ← i;
        makespan_parcial ← calcular_makespan_parcial();
        if makespan_parcial <min_makespan then
            min_makespan ← makespan_parcial;
            tarea_minima ← i;
    orden(t) ← tarea_minima;
    Eliminar de tareas_no_asignadas
    for j ∈ maquinas do
        Establecer tiempo de inicio y fin;
        Establecer recursos;
```

El algoritmo se basa en recorrer todas las tareas disponibles introduciendo cada tarea como solución en la secuencia y calculando el *makespan* parcial que resulta de introducirla. De esta manera, siempre se va tratando de minimizar el *makespan* total. En nuestro problema puede que no sea tan efectivo, ya que después de la fase de construcción, se comprueban los recursos, de tal manera que la solución que inicialmente es más eficiente, puede verse muy afectada por los retrasos que originan la disponibilidad de los recursos.

También debemos modificar el algoritmo *GRASP* para que utilice la heurística basada en NEH en la fase de construcción de una solución inicial (*Greedy Aleatorizado*), en vez que con el otro criterio. Este algoritmo *greedy* aleatorizado se vería modificado de la siguiente manera que se presenta en el Algoritmo 7.

Algoritmo 7: Greedy aleatorizado con criterio basado en NEH.

```
for t ∈ tareas do
    RCL vacía;
    Añadir a tareas_no_asignadas las tareas que no están en la solución;
    for h ∈ 1 : k do
        for i ∈ tareas_no_asignadas and i ∉ RCL do
            Obtener la tarea con menor makespan_parcial;
        Añadir a la RCL la tarea con menor makespan_parcial;
        Eliminar de tareas_no_asignadas;
    Obtener una tarea aleatoria de la RCL;
    orden(t) ← tarea_aleatoria;
    Calcular tiempos de inicio y fin;
```

Donde se observa que en cada iteración, partiendo de una RCL vacía, se van añadiendo los mejores trabajos en función del menor *makespan* parcial, obtiendo una lista de candidatos de tamaño k. Una vez obtenida la lista, se selecciona aleatoriamente una de las tareas de la lista y se añade a la secuencia, calculando el tiempo de inicio, fin y la cantidad de recursos necesarios en cada instante de tiempo.

Una vez modificados los métodos necesarios que se van a utilizar, se pasa a realizar el mismo procedimiento de experimentos computacionales que en el apartado anterior, obteniendo los resultados de ejecución y *makespan* para cada uno de los algoritmos.

A continuación se presentan los resultados obtenidos para cada uno de los algoritmos propuestos, así como el tiempo de ejecución del algoritmo y el *makespan* obtenido en las Tablas 6.3 y 6.4.

Instancia	**Greedy**	**Greedy Aleatorizado 500**	**Greedy + BL**	**GRASP 500**
6x2_inst_1	0.013	11.06	0.037	14.623
6x2_inst_2	0.015	11.955	0.033	17.707
8x3_inst_1	0.048	27.018	0.08	43.678
8x3_inst_2	0.044	21.295	0.068	37.598
10x4_inst_1	0.082	37.114	0.123	82.626
10x4_inst_2	0.071	40.045	0.114	74.258
20x4_inst_1	0.199	98.016	0.558	328.467
30x4_inst_1	0.405	176.481	1.308	847.996
50x4_inst_1	0.99	434.272	3.076	2890.371

Tabla 6.3: Tiempos de ejecución obtenidos para el criterio NEH.

Instancia	**Greedy**	**Greedy Aleatorizado 500**	**Greedy + BL**	**GRASP 500**
6x2_inst_1	177	136	161	136
6x2_inst_2	248	170	193	170
8x3_inst_1	519	495	512	493
8x3_inst_2	372	274	304	275
10x4_inst_1	548	513	537	518
10x4_inst_2	644	557	590	545
20x4_inst_1	1168	1099	1174	1077
30x4_inst_1	1673	1577	1656	1536
50x4_inst_1	2635	2475	2495	2391

Tabla 6.4: *Makespan* obtenidos para el criterio NEH.

Como podemos observar en las tablas de resultados, podemos obtener conclusiones parecidas a las que comentabamos anteriormente con el primer criterio. El algoritmo que mejores resultados nos proporciona, por lo general ha sido el algoritmo *GRASP* con 500 iteraciones, obteniendo un *makespan* más bajo en la mayoría de los casos, aunque evidentemente, sea el que más tiempo requiera para encontrar dicha solución.

Si prestamos atención en las instancias más pequeñas, observamos como el *Greedy aleatorizado* obtiene mejores resultados que el *Greedy* básico, ya que nos podemos ver beneficiados por la aleatorización al elegir una tarea de entre las de la RCL, dando un mejor resultado. No obstante, esto se suprime en las instancias más grandes, donde el algoritmo *Greedy* básico obtiene mejores resultados que el *Greedy aleatorizado*, donde el azar ya no es tan determinando al aumentar el número de permutaciones posibles.

Por otro lado, podemos llegar a pensar que el algoritmo *Greedy + BL* debería obtener siempre mejores resultados que el algoritmo *Greedy* ya que estamos introduciendo una fase más con el objetivo de mejorar la solución inicial que nos proporciona el algoritmo *Greedy*. Sin embargo, esto no es del todo así, ya que aunque si es cierto que debería mejorar la solución inicial que nos proporciona el algoritmo *Greedy*, debemos tener en cuenta además, la fase de reparación de recursos, que puede empeorar aún más que la situación inicial.

6.3. Análisis comparativo de los métodos

Para obtener una mejor visión de los resultados obtenidos, vamos a realizar un análisis de los resultados obtenidos en las tablas anteriores, comparando los resultados de cada uno de los algoritmos.

Para comparar las heurísticas, vamos a hacer uso del **RPD** *Relative Percentage Deviation* que nos permite calcular el porcentaje de la distancia relativa de cada uno de los métodos respecto al mejor método obtenido en cada una de las instancias. Este método es conveniente puesto que son distancias relativas y evitamos errores al no depender de unidades.

$$RPD = \frac{(makespan_{alg} - makespan_{best})}{makespan_{best}} \cdot 100 \tag{6.1}$$

Donde $makespan_{alg}$ es el *makespan* obtenido por el algoritmo que estamos analizando y $makespan_{best}$ es el *makespan* obtenido por el mejor algoritmo en la misma instancia.

Seguido de calcular los RPD, realizamos un Análisis de la Varianza ANOVA con un factor, donde la variable respuesta es el *RPD* y el factor es el algoritmo utilizado. Tras ajustar el modelo y observar que no existe igualdad de medias,tenemos que asegurar los resultados. Para ello, realizamos los diferentes test post-hoc para comprobar si existen diferencias significativas entre los algoritmos propuestos.

Primero tenemos que comprobar si los residuos del modelo ajustado siguen una distribución normal y si existe homocedasticidad, es decir, igualdad de varianzas. Si esto fuera así, se aplicaría un test paramétrico como es el test de Tukey para comprobar diferencias. En caso contrario, aplicaríamos un test no paramétrico como el test de Kruskal-Wallis. Un test paramétrico asume que los datos siguen una distribución específica y se basan en parámetros poblacionalmente conocidos. Los tests no paramétricos, por el contrario, no asumen ninguna distribución específica y se basan en rangos o posiciones de los datos con el objetivo de dar más flexibilidad. Para los tests de normalidad y homocedasticidad usaremos el test de Shapiro-Wilk y Levene respectivamente.

6.3.1. Comparación de métodos para el criterio basado en menor tiempo de ejecución en máquina 1

Lo primero que debemos hacer para comparar los métodos es calcular el **RPD** para cada uno de los algoritmos y cada una de las instancias. Para ello, obtenemos una tabla con todos los RPD calculados por instancia, como podemos ver en la Tabla 6.5.

Instancia	Greedy	Greedy Aleatorizado 500	Greedy + BL	GRASP 500
6x2_inst_1	13.97	0.00	16.18	0.00
6x2_inst_2	0.48	0.00	0.48	17.14
8x3_inst_1	1.43	0.00	4.10	0.00
8x3_inst_2	15.64	0.36	7.27	0.00
10x4_inst_1	6.40	0.00	15.80	3.60
10x4_inst_2	7.04	0.00	7.04	0.00
20x4_inst_1	6.62	0.92	5.15	0.00
30x4_inst_1	5.98	0.00	7.03	1.97
50x4_inst_1	4.31	0.41	1.20	0.00
100x3_inst_1	3.92	2.18	1.46	0.00

Tabla 6.5: RPD obtenidos para el criterio de menor tiempo de ejecución en la primera máquina.

Una vez calculados, modificamos el formato de la tabla para obtener un conjunto de datos longitudinal, donde cada fila corresponde a un algoritmo y una instancia y ajustamos el modelo ANOVA a través del lenguaje de programación R.

```
            Df Sum Sq Mean Sq F value  Pr(>F)
Algoritmo    3  293.0   97.65   4.654 0.00753 **
Residuals   36  755.3   20.98
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Observamos como rechazamos la hipótesis nula de igualdad de medias, por lo que podemos concluir que existen diferencias significativas entre los algoritmos propuestos. Podemos realizar un análisis post-hoc para comprobar qué algoritmos son los que presentan diferencias significativas entre ellos. Previamente, debemos comprobar si usar un test paramétrico o no. Para ello, comprobamos la normalidad de los residuos y la homogeneidad de la varianza a través de los tests de Shapiro-Wilk y Levene respectivamente.

```
Shapiro-Wilk normality test

data:  residuals(anova_result)
W = 0.81201, p-value = 1.218e-05

Levene's Test for Homogeneity of Variance (center = median)
       Df F value Pr(>F)
group  3  1.9495 0.1391
    36
```

El test de Shapiro-Wilk nos rechaza la hipótesis nula de normalidad de los residuos. Aunque el test de Levene no rechaza la hipótesis nula de homogeneidad de varianza, no podemos aplicar un test parámetrico como el test de Tukey. Por lo tanto, reemplazamos por un test no paramétrico como el test de Kruskal-Wallis.

```
Kruskal-Wallis rank sum test

data:  RPD by Algoritmo
Kruskal-Wallis chi-squared = 20.336, df = 3, p-value = 0.0001446
```

El test de Kruskal-Wallis nos indica que existen diferencias significativas entre los algoritmos propuestos. Podemos realizar un análisis post-hoc a través de métodos numéricos como por ejemplo el test de Bonferroni-Dunn o el test de Nemenyi que se basan en el cálculo de rangos entre métodos a través de una distancia crítica. Sin embargo, vamos a comprobar las diferencias gráficamente. Estas diferencias se pueden observar en la Tabla 6.1, donde se presentan los boxplots de los RPD obtenidos para cada uno de los algoritmos.

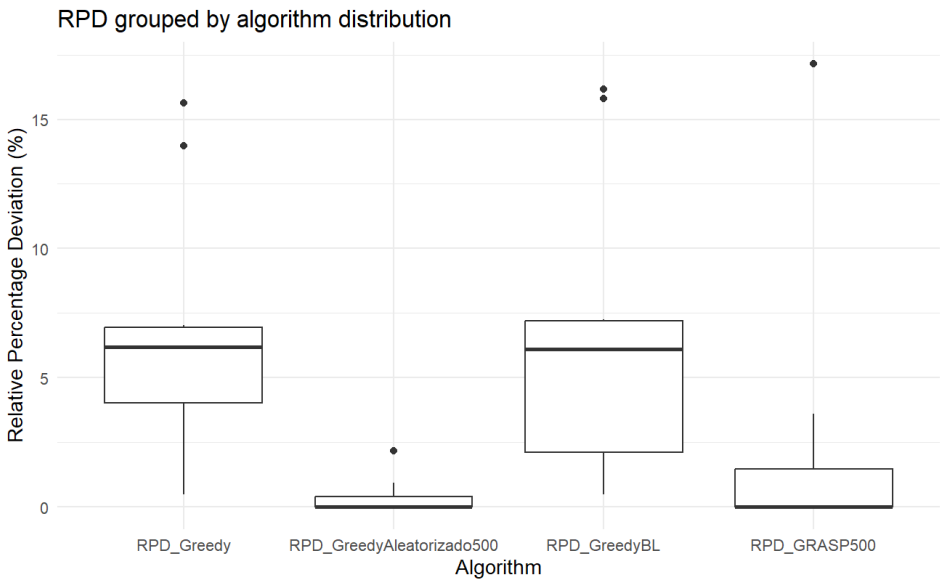

Figura 6.1: Boxplot de los RPD obtenidos para el criterio de menor tiempo de ejecución en máquina 1.

El gráfico de cajas de la Figura 6.1 nos muestra las diferencias entre algoritmos. Por ejemplo podemos observar como los algoritmos *Greedy Aleatorizado* y *GRASP* son los que mejores resultados, incluso para este tipo de instancias, sobre todo en las más pequeñas, actua mejor el algoritmo *Greedy Aleatorizado*. En realidad, aunque el banco de pruebas en este análisis es pequeño, es más lógico pensar que el algoritmo *GRASP* es la mejor opción, ya que es el más complejo y el que más probabilidades tiene de conducir a un óptimo.

Sin embargo, el algoritmo *Greedy* básico y el *Greedy + BL* no parecen ser una buena opción, ya que aunque en instancias pequeñas obtienen buenos resultados, en instancias más grandes no son capaces soluciones óptimas que se acerquen a los algoritmos que contienen aleatorización.

6.3.2. Comparación de métodos para el criterio basado en NEH

Para comenzar con el análisis, obtenemos una tabla con todos los RPD calculados por instancia, como podemos ver en la Tabla 6.6.

Instancia	Greedy	Greedy Aleatorizado 500	Greedy + BL	GRASP 500
6x2_inst_1	30.15	0.00	18.38	0.00
6x2_inst_2	45.88	0.00	13.53	0.00
8x3_inst_1	5.27	0.41	3.85	0.00
8x3_inst_2	35.77	0.00	10.95	0.36
10x4_inst_1	6.82	0.00	4.68	0.97
10x4_inst_2	18.17	2.20	8.26	0.00
20x4_inst_1	8.45	2.04	9.01	0.00
30x4_inst_1	8.92	2.67	7.81	0.00
50x4_inst_1	10.20	3.51	4.35	0.00

Tabla 6.6: RPD obtenidos para el criterio NEH.

Como en el caso anterior, ajustamos con estos datos un modelo ANOVA para comprobar si existen diferencias significativas entre los algoritmos propuestos. Esto se realiza por medio del lenguaje de programación R. Para ello ha sido necesario modificar del mismo modo, el formato de la Tabla 6.6 para poder obtener una columna como variable respuesta *RPD* y crear la variable categórica 'Algoritmo'.

Los resultados que obtenemos del ANOVA son los siguientes:

```
            Df Sum Sq Mean Sq F value   Pr(>F)
Algoritmo    3   2020   673.5   11.03 3.93e-05 ***
Residuals   32   1953    61.0
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Como observamos en la salida que nos proporciona el test F del modelo ANOVA, el p-valor asociado rechaza la hipótesis nula de igualdad de medias, por lo que podemos concluir que existen diferencias significativas entre los algoritmos propuestos.

Podemos realizar un análisis post-hoc para comprobar qué algoritmos son los que presentan diferencias significativas entre ellos. Sin embargo, antes de realizar dicho test, necesitamos comprobar si usar un test paramétrico o no. Esto lo conseguimos, comprobando la normalidad de los residuos y la homogeneidad de la varianza.

```
Shapiro-Wilk normality test

data:  residuals(anova_result)
W = 0.82941, p-value = 6.616e-05

Levene's Test for Homogeneity of Variance (center = median)
    Df F value   Pr(>F)
group  3  5.1356 0.005175 **
    32
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Como podemos observar, el p-valor es menor que 0.05 en ambos tests, por lo que rechazamos la hipótesis nula de normalidad de los residuos y homogeneidad de la varianza. Por lo tanto, no podemos aplicar el test paramétrico de Tukey. Así pues, en el caso contrario, nos decantaremos por un test no paramétrico al igual que en el anterior caso, el test de Kruskal-Wallis, que no requiere una distribución concreta sino que utiliza los rangos de los métodos.

```
Kruskal-Wallis rank sum test

data:  RPD by Algoritmo
Kruskal-Wallis chi-squared = 28.228, df = 3, p-value = 3.254e-06
```

El test de Kruskal-Wallis también nos rechaza la hipótesis de igualdad de medianas entre grupos, por lo que podemos concluir que existen diferencias significativas entre los algoritmos propuestos. Para una mayor visualización de los resultados, en la Figura 6.2 se presenta un boxplot con los resultados obtenidos, donde podemos observar la diferencia entre los algoritmos propuestos.

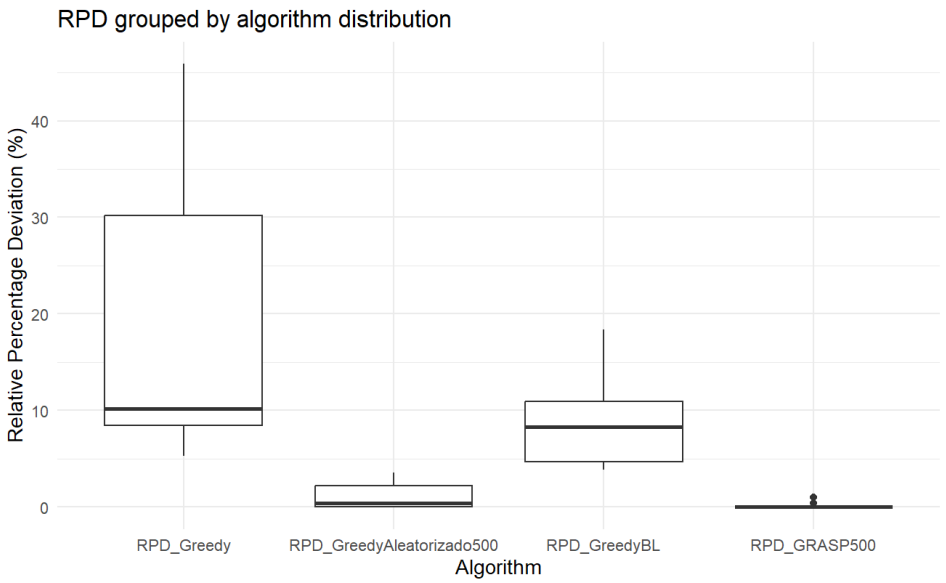

Figura 6.2: Boxplot de los RPD obtenidos para el criterio NEH.

A la vista del gráfico de cajas, podemos concluir cómo el algoritmo *GRASP* es el que mejores resultados nos proporciona, ya que es el que menos *RPD* presenta junto con el *Greedy Aleatorizado*. Por otro lado, el algoritmo *Greedy* es el que peores resultados nos proporciona, ya que es el que más *RPD* presenta.

Podemos incluso mencionar que el algoritmo con criterio aleatorizado puede llegar a ser el mejor debido a su menor tiempo de ejecución en proporción con cuánto peor es con respecto al *GRASP*. Sin embargo, el rango de pruebas del análisis es limitado y no podemos asegurar dicha afirmación, dependiendo de cada situación que tratemos.

Capítulo 7

Conclusiones y tabajo futuro

El Flowshop Scheduling Problem es un problema de optimización complejo pero que demuestra que podemos obtener soluciones óptimas para instancias pequeñas, pero para instancias más grande puede no ser posible. En este Trabajo de Fin de Grado hemos diseñado, implementado y probado diferentes variantes del algoritmo *greedy* para obtener las mejores soluciones según la instancia.

Los experimentos y análisis computacionales realizados nos demuestran que la aleatorización es un factor sumamente importante en la optimización y que nos ayuda en gran medida a obtener resultados realmente buenos cuando tenemos problemas de este tipo. La aleatorización nos permite explorar el espacio de soluciones de una manera más eficiente y nos ayuda a evitar caer en óptimos locales.

Además la elección de los criterios de selección de trabajos también puede ser un factor determinante en la calidad de las soluciones, ya que aunque no existe un criterio absoluto que nos garantice una mejor solución, dependiendo del problema que estemos resolviendo, algunos criterios pueden ser más efectivos que otros. En nuestro caso, hemos visto como el criterio de mínimo tiempo de ejecución en máquina 1 parece obtener mejores resultados que el algoritmo basado en NEH, aunque nuestro objetivo con este trabajo no era este.

Podemos concluir afirmando que los algoritmos GRASP y *greedy* aleatorizado con repetición son buenas opciones para resolver el Flowshop Scheduling Problem, gracias a la implementación de la aleatorización que mencionamos anteriormente y que nos ayuda a obtener mejores soluciones a coste de más repeticiones. También mencionamos que la aleatorización y la iteración nos producen un mayor coste computacional como es evidente, pero puede ser un coste a asumir dependiendo de la situación y el problema que estemos tratando, por ello es importante tener en cuenta el contexto y los requisitos del problema a la hora de elegir un algoritmo de optimización.

Al tratar en este trabajo un tema de optimización complejo en el cual se podrían probar infinidad de soluciones, se propone a continuación una serie de posibles líneas de trabajo futuras que podrían llevarse a cabo para mejorar el trabajo realizado:

- Búsqueda de otros criterios de selección de trabajos que nos permitan obtener los mejores resultados posibles.
- Probar la implementación por tiempo del algoritmo GRASP en vez de número de iteraciones, o incluso por tiempo sin encontrar una mejor solución
- Implementación de la fase de reparación después de cada fase (construcción, búsqueda local) y observar su comportamiento en los métodos utilizados.
- Aleatorizar la fase de reparación del algoritmo propuesto. Esto podría aplicarse a la hora de aplicar un retraso de tareas en la secuencia, insertando de nuevo de manera aleatoria entre varias opciones.
- Mayor optimización del código para mejorar el tiempo de ejecución, ya que en este trabajo se ha priorizado la claridad del problema y su compresión.

Esta selección de líneas de trabajo futuras serían interesantes de llevar a cabo con el objetivo de mejorar las soluciones en el Flowshop Scheduling Problem y obtener mejores resultados en el tiempo de ejecución. Además, se podrían explorar otras variantes del algoritmo greedy y su combinación con otros métodos de optimización.

Bibliografía

[1] Wikipedia. Flow-shop scheduling. https://en.wikipedia.org/wiki/Flow-shop_scheduling. Accessed: 2025-02-27.

[2] Quan-Ke Pan, Rubén Ruiz. A comprehensive review and evaluation of permutation flowshop heuristics to minimize flowtime. https://www.sciencedirect.com/science/article/abs/pii/S0305054812001347. Accessed: 2025-02-27.

[3] Jose M. Framinan, Rainer Leisten, and Rafael Ruiz-Usano. Comparison of heuristics for flowtime minimisation in permutation flowshops. https://www.sciencedirect.com/science/article/pii/S0305054803003198. Accessed: 2025-04-10.

[4] J.N.D. Gupta. Heuristic algorithms for multistage flowshop scheduling problem. https://www.tandfonline.com/doi/abs/10.1080/05695557208974823. Accessed: 2025-04-10.

[5] Kenneth Steiglitz Martin J. Krone. Heuristic-programming solution of a flowshop-scheduling problem. https://pubsonline.informs.org/doi/10.1287/opre.22.3.629. Accessed: 2025-04-10.

[6] Fumio Hashimoto Shigeji Miyazaki, Noriyuki Nishiyama. Solving the mean residence time scheduling problem by the adjacent two-job exchange method. https://www.jstage.jst.go.jp/article/jorsj/21/2/21_KJ00002432742/_article/-char/ja/. Accessed: 2025-04-10.

[7] Inyong Ham Muhammad Nawaz, E Emory Enscore Jr. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. https://www.sciencedirect.com/science/article/pii/0305048383900889. Accessed: 2025-04-10.

[8] Colin R Reeves Jiyin Liu. Constructive and composite heuristic solutions to the scheduling problem. https://www.sciencedirect.com/science/article/pii/S0377221700001375. Accessed: 2025-04-10.

[9] Tariq Aldowaisan Ali Allahverdi. New heuristics to minimize total completion time in m-machine flowshops. https://www.sciencedirect.com/science/article/pii/S0925527301002286. Accessed: 2025-04-10.

[10] Dong-Soon Yim Hoon-Shik Woo. A heuristic algorithm for mean flowtime objective in flowshop scheduling. https://www.sciencedirect.com/science/article/pii/S0305054897000506. Accessed: 2025-04-10.

[11] Hans Ziegler Chandrasekharan Rajendran. An efficient heuristic for scheduling in a flowshop to minimize total weighted flowtime of jobs. https://www.sciencedirect.com/science/article/pii/S0377221796002731. Accessed: 2025-04-10.

[12] Czesław Smutnicki Eugeniusz Nowicki. A fast tabu search algorithm for the permutation flow-shop problem. https://www.sciencedirect.com/science/article/pii/0377221795000372. Accessed: 2025-04-10.

[13] CN Potts IH Osman. Simulated annealing for permutation flow-shop scheduling. https://www.sciencedirect.com/science/article/pii/0305048389900595. Accessed: 2025-04-10.

[14] Colin R. Reeves. A genetic algorithm for flowshop sequencing. https://www.sciencedirect.com/science/article/pii/0305054893E0014K. Accessed: 2025-04-10.

[15] Pedro Alfaro Fernández. Inteligencia computacional en la programación de la producción con recursos adicionales. https://riunet.upv.es/entities/publication/20857ef1-02c6-48c6-9432-75ca84a2cc42. Accessed: 2025-02-13.

[16] Pedro Alfaro Fernández. Inteligencia computacional en la programación de la producción con recursos adicionales. https://riunet.upv.es/handle/10251/198891. Accessed: 2025-02-13.

[17] Leonidas S. Pitsoulis and Mauricio G. C. Resende. Greedy randomized adaptive search procedures. https://mauricio.resende.info/doc/grasp-hao.pdf. Accessed: 2025-03-28.