Universidad de Valladolid

FACULTAD DE CIENCIAS

TRABAJO FIN DE GRADO

Grado en Estadística

ALGORITMOS DE OPTIMIZACIÓN PARA EL PROBLEMA DISTRIBUTED JOB-SHOP

Autor: Alejandro Miñambres Mateos
Tutores: Juan Camilo Yepes Borrero &
Jesús Alberto Tapia García
Año 2024/2025

Resumen

Este trabajo se centra en el estudio y la resolución del *Distributed Job-Shop Scheduling Problem* (DJSSP), una extensión del clásico problema de planificación de trabajos (JSSP) considerando múltiples fábricas heterogéneas. El objetivo es asignar eficientemente los trabajos en las diferentes fábricas, para minimizar el *makespan* (tiempo total de la producción). El trabajo incluye una detallada formulación matemática y el desarrollo de dos enfoques de solución: heurística Greedy con tres criterios diferentes y una metaheurística GRASP, basada en los mismos criterios. Para la comparación de los resultados se aplican diferentes técnicas estadísticas como el test de Friedman, ANOVA de medidas repetidas y el test de Nemenyi, evaluando así el rendimiento de los algoritmos propuestos.

Palabras clave: Distributed Job-Shop Scheduling Problem, algoritmos, heurísticas, Greedy, GRASP, makespan, planificación, optimización.

Abstract

This work focuses on the study and resolution of the Distributed Job-Shop Scheduling Problem (DJSSP), an extension of the classic Job-Shop Scheduling Problem (JSSP) that involves multiple heterogeneous factories. The main objective is to efficiently assign jobs across the different factories in order to minimize the makespan (total production time). The study includes a detailed mathematical formulation of the problem and develops two solution approaches: a Greedy heuristic with three different criteria, and a GRASP metaheuristic based on the same criteria. To compare the results, several statistical techniques are applied, including the Friedman test, repeated measures ANOVA, and the Nemenyi test, thereby evaluating the performance of the proposed algorithms.

Keywords: Distributed Job-Shop Scheduling Problem, algorithms, heuristics, Greedy, GRASP, makespan, scheduling, optimization.

Índice

Índice de tablas

1. Introducción

El incremento de la competitividad y la exigencia en el mercado es constante, y para las compañías es crucial destacar y distinguirse. Por esa razón, la investigación operativa tiene una gran importancia, ya que permite optimizar procesos y mejorar la toma de decisiones, lo que es imprescindible para destacar en un mercado tan competitivo. Dentro de la investigación operativa surgen una gran cantidad de problemas, algunos comúnmente conocidos como *El problema de asignación y el transporte*. Estos problemas tienen a su vez una gran variedad de opciones y restricciones, lo que añade una gran complejidad.

Este trabajo está centrado en problemas *Scheduling*, más concretamente en el problema conocido como *Distributed Job-Shop Scheduling* (DJSSP). Este tipo de problema tiene mucha importancia en el mundo industrial, ya que consiste en organizar de forma eficiente un conjunto de trabajos o tareas que deben ser procesados en diferentes máquinas. Cada trabajo necesita pasar por varias máquinas en un orden específico, lo que implica realizar una planificación cuidadosa.

A este nivel de complejidad se le añade otra característica adicional, la distribución de los trabajos en diferentes fábricas. Esto complica la planificación, ya que no solo hay que decidir el orden de ejecución de los trabajos en las máquinas, sino también a qué fábrica se asignará cada uno de ellos.

Por último, el problema que se aborda en este trabajo es de fábricas heterogéneas; esto significa que la secuencia necesaria de máquinas y tiempos para completar un mismo trabajo puede variar en función de la fábrica a la que se asigne el trabajo. Esta variabilidad aumenta considerablemente la dificultad del problema.

El objetivo final es minimizar el tiempo total necesario para completar todos los trabajos, lo que se conoce como *makespan*.

1.1. Motivación

La motivación principal es encontrar buenas soluciones al problema *Distributed Job-Shop Scheduling*, aplicando diferentes algoritmos para poder minimizar la función objetivo. Debido a la complejidad del problema, no es posible obtener una solución óptima. Este tipo de problemas se denominan NP-Hard; la complejidad se incrementa en gran medida de forma polinómica según aumenta el tamaño del problema, por lo que se propondrán algoritmos y metaheurísticas con el objetivo de comparar los resultados y obtener conclusiones sobre cuáles funcionan mejor.

1.2. Objetivos

1.2.1. Objetivo general

Principalmente, el objetivo general es comprender, comparar, analizar y evaluar diferentes algoritmos y metaheurísticas para abordar el problema *Distributed Job-Shop Scheduling* con el fin de encontrar soluciones eficientes y aproximar la solución óptima.

El problema DJSSP es una extensión del clásico problema JSSP (Job-Shop-Scheduling Problem). Se parte de un conjunto de trabajos que tienen una secuencia de operaciones que deben realizarse en un orden específico sobre un conjunto de máquinas. La complejidad aumenta en el caso distribuido, ya que esas operaciones pueden realizarse en varias fábricas diferentes y no necesariamente las fábricas tienen las mismas máquinas. Además, se añade complejidad al problema considerando un entorno de fábricas heterogéneas, lo que implica que el orden y la duración de las operaciones pueden variar dependiendo de la fábrica a la que se asigne el trabajo.

Para abordar de forma correcta el problema, primero se estudiarán los fundamentos teóricos del problema y las características que lo definen. A continuación, se implementarán distintos enfoques computacionales para luego comparar el desempeño de cada uno en los distintos escenarios de prueba.

1.2.2. Objetivos específicos

Entender las bases teóricas de los problemas de planificación de tareas conocidos como Scheduling Problems, incluyendo versiones clásicas como el JSSP (Job-Shop Scheduling Problem), para luego examinar más a fondo la versión distribuida DJSSP (Distributed Job-Shop Scheduling Problem); esta variante incorpora múltiples fábricas y una mayor complejidad.

Otro de los desafíos consiste en analizar en profundidad el problema DJSSP desde una perspectiva computacional, explorando y describiendo diferentes algoritmos y metaheurísticas que se emplean para su resolución aproximada.

Investigar, implementar y analizar los diferentes algoritmos y metaheurísticas que sirven para resolver el DJSSP, prestando atención a las ventajas, limitaciones y eficiencia.

Un objetivo importante es comparar los diferentes algoritmos propuestos mediante diversas simulaciones para poder evaluar su eficiencia en la búsqueda de la mejor solución.

Finalmente, elaborar conclusiones y sugerencias basadas en los resultados obtenidos, planteando posibles mejoras en algoritmos o nuevas estrategias para abordar el problema DJSSP.

1.3. Contenidos

El trabajo se organiza de la siguiente manera:

- En el capítulo 2 se introduce el contexto general de los problemas de planificación de la producción, destacando su importancia en el ámbito industrial.
- En el capítulo 3 se presenta el problema Distributed Job-Shop Scheduling Problems (DJSSP), incluyendo su definición, características, formulación matemática y un ejemplo ilustrativo.
- En el capítulo 4 se recoge la revisión bibliográfica relacionada con los problemas de planificación y el estado del arte del DJSSP.
- En el capítulo 5 se describen las técnicas estadísticas empleadas para comparar el rendimiento de los algoritmos desarrollados.
- En el capítulo 6 se describen las instancias utilizadas para realizar las pruebas experimentales.
- En el capítulo 7 se explican los diferentes algoritmos propuestos. Una heurística Greedy con tres criterios y la metaheurística GRASP con los mismos tres criterios.
- En el capítulo 8 se realiza la comparación entre la heurística Greedy y la metaheurística GRASP en función del rendimiento obtenido.
- En el capítulo 9 se presentan las conclusiones y posibles lineas futuras de investigación y de mejora.

2. Problemas de planificación de la producción

La planificación de la producción es un área dentro de la optimización cuyo objetivo es asignar tareas a recursos de manera eficiente. Este tipo de problemas son conocidos como *Scheduling Problems*. Se basan en organizar un conjunto de actividades que compiten por el uso de un conjunto limitado de recursos. Se trata de optimizar una o varias funciones objetivo, las cuales están sujetas a unas restricciones [1].

Los *Scheduling Problems* cumplirán 70 años en 2026.

En Kelley y Walker se introduce el primero de estos problemas, comenzaron a desarrollar algoritmos que publicarían en 1959 en su primer artículo sobre las rutas críticas [2].

Estos problemas generan un gran interés, tanto a nivel económico como a nivel académico; por esa razón, han sido objeto de investigación intensiva durante las últimas décadas por parte de las comunidades de Investigación Operativa [1].

Sin embargo, estos conceptos no son nuevos; ejemplos como la construcción de las pirámides hace más de 3000 años o los ferrocarriles transcontinentales no podrían haberse realizado sin comprender las actividades y la secuenciación, es decir, los primeros *Scheduling Problems* [3].

Se ha demostrado que la mayoría de estos problemas de este tipo son NP-Hard [4]. Esto hace necesario proponer algoritmos que se aproximen a la solución óptima.

El *Job-Shop Scheduling Problem* (JSSP) es un subtipo de *Scheduling Problems* y constituye uno de los retos más analizados en el ámbito de la optimización y la investigación operativa. Este problema es especialmente importante en el sector industrial y en la fabricación, donde diversos trabajos o tareas deben realizarse en un conjunto de máquinas siguiendo un orden específico de operaciones.

El JSSP es esencial para manufacturas, empresas logísticas y de producción, ya que favorece la mejora de la eficiencia operativa, disminuyendo tiempos de espera, lo que reduce gastos y optimiza el uso de los recursos. Su uso práctico abarca fábricas, talleres mecánicos, líneas de montaje y centros de distribución, donde una planificación adecuada puede resultar en una notable ventaja competitiva [5].

Además, dado que se trata de un problema NP-hard, el JSSP tiene también una gran relevancia en el ámbito académico y científico, ya que actúa como base para la exploración de nuevos algoritmos, técnicas de optimización y métodos inteligentes como heurísticas, metaheurísticas o algoritmos evolutivos.

El problema consiste en un conjunto de trabajos, los cuales deben ser procesados en máquinas, teniendo cada trabajo una secuencia de tareas que deben ser realizadas en un orden particular. Pese a que pueden plantearse diferentes objetivos, el

más común suele ser minimizar el tiempo que tarda en finalizar todos los trabajos (*makespan*).

3. Distributed Job-Shop Scheduling Problems

El *Distributed Job-Shop Scheduling Problem* (DJSSP) es una extensión del JSSP. El problema se plantea inicialmente como el JSSP, solo que las tareas están distribuidas en diferentes lugares (fábricas).

Por lo tanto, el DJSSP puede describirse como un conjunto de trabajos, que son asignados a un conjunto de fábricas y procesados en un conjunto de máquinas.

Cada trabajo tarda un tiempo único en ser procesado en cada máquina. Además, hay un conjunto de restricciones las cuales hay que cumplir:

- Todos los trabajos y máquinas están disponibles desde el inicio para su procesamiento.
- No se pueden cancelar los trabajos durante su procesamiento.
- Todos los trabajos tienen el mismo peso, es decir, tienen la misma importancia.
- Los trabajos no tienen fecha de vencimiento (caducidad).
- No existe prioridad entre los distintos trabajos.
- Los trabajos son independientes entre si.
- Cada trabajo tiene una ruta de procesamiento única.
- Cada máquina puede procesar cualquier trabajo. Es decir, las máquinas no tienen capacidades específicas.
- Cada máquina solo puede procesar un único trabajo a la vez.
- Se permite la espera entre procesamiento.
- Un trabajo no visita la misma máquina varias veces, es decir, no se considera la re-entrada.
- No se consideran los tiempos de configuración de las máquinas.
- No existen trabajos de emergencia que haya que realizar con urgencia.
- Se supone que la capacidad del almacén es limitada.

Una vez vistas todas estas restricciones del JSSP, pero que se extienden a DJSSP, hay que destacar que se pueden tener fábricas homogéneas o heterogéneas. La diferencia se entiende con un ejemplo.

Suponemos que hay un trabajo J_1 el cual necesita ser procesado en una máquina M_1.

En fábricas homogéneas, da igual si M_1 está en una fábrica F_1 o en otra fábrica F_2, el tiempo p de procesamiento será el mismo.

Sin embargo, en fábricas heterogéneas, el tiempo p sería diferente si se procesa en F_1 que si se procesa en F_2 y, además, en F_2 podría tener un orden diferente de procesamiento, es decir, que en F_1 primero se necesita M_1 y luego M_2, pero en F_2 podría ser al revés.

A partir de ahora se supone que las fábricas son heterogéneas, lo que se adecúa más a la realidad, ya que no todas las fábricas son iguales, hay diferentes máquinas, y no todas las fábricas son igual de eficientes; además, el caso de las fábricas homogéneas es un caso particular del caso de fábricas heterogéneas.

Además de las restricciones vistas anteriormente del JSSP, hay que tener en cuenta otras restricciones propias del problema DJSSP.

- Existen diferentes fábricas, cada una sus propias máquinas.
- No está permitido mover trabajos entre fábricas, por lo que no se tiene en cuenta el tiempo de transporte entre fábricas.
- Las fábricas son heterogéneas: las máquinas y capacidades varían entre fábricas.

Para resolver el problema DJSSP hay que tener en cuenta todas las restricciones. El DJSSP añade las restricciones de asignación y heterogeneidad, lo que lo convierte en una extensión más realista, pero también mucho más difícil que el JSSP.

3.1. Formulación matemática del modelo de programación lineal

Para entender mejor el problema, se formula y explica el modelo matemático.

El DJSSP puede describirse como un conjunto de trabajos $J = \{J_1, J_2, \ldots, J_n\}$, los cuales tienen que ser asignados a un conjunto de fábricas $F = \{F_1, F_2, \ldots, F_f\}$ y procesados en un conjunto de máquinas $M = \{M_1, M_2, \ldots, M_m\}$ [6].

El modelo matemático viene formulado en el *paper* [6], sin embargo, en esa formulación se consideran fábricas homogéneas. Como este trabajo es con fábricas heterogéneas, se reformula el modelo. Además, en ese mismo *paper* [6] se han detectado por el autor de este TFG algunos pequeños errores en la formulación matemática del modelo, que una vez corregidos conducen a la siguiente formulación.

Para empezar, se declaran las siguientes variables:

n : El número de trabajos

m : El número de máquinas

f : El número de fábricas

p_{jik} : El tiempo de procesamiento del trabajo j en la máquina i, en la fábrica k

$a_{j,i_1,i_2,k}$: Si la máquina i_1 es usada inmediatamente después de la máquina i_2 en el procesamiento del trabajo j en la fábrica k; $a_{j,i_1,i_2,k} = 1$ en otro caso $a_{j,i_1,i_2,k} = 0$

P : Un número positivo muy grande.

$X_{j_1,j_2,i,k}$: Si el trabajo j_2 es procesado después del trabajo j_1 en la máquina i en la fábrica k; $X_{j_1,j_2,i,k} = 1$, en otro caso $X_{j_1,j_2,i,k} = 0$

$Y_{j,k}$: Si el trabajo j es procesado en la fábrica k; $Y_{j,k} = 1$, en otro caso $Y_{j,k} = 0$

$C_{j,i,k}$: El tiempo de finalización del trabajo j en la máquina i en la fábrica k.

$C_{\text{máx}}$: El tiempo total de procesamiento de los trabajos en todas las máquinas.

Como se habrá podido observar, para facilitar la interpretación, se intentarán usar los mismos subíndices, de tal forma que los trabajos siempre serán el subíndice j, las máquinas siempre serán el subíndice i y las fábricas siempre serán el subíndice k.

El objetivo principal es minimizar el tiempo total.

$$\min C_{\text{máx}} \tag{1}$$

La ecuación 1 es la función objetivo, minimizar el *makespan* global de todas las fábricas.

$$\sum_{r=1}^{f} Y_{jr} = 1, \quad \forall j \tag{2}$$

La restricción 2 garantiza que todos los procesamientos de un trabajo estén asignados en la misma fábrica.

$$C_{jik} \geq p_{jik}, \quad \forall j, i, k \tag{3}$$

La restricción 3 garantiza que el tiempo de finalización de cualquier operación sea mayor que su tiempo de procesamiento.

$$C_{ji_1k} \geq C_{ji_2k} + p_{jik}, \quad \forall j, k, i_2, i_1 \neq i_2 \mid a_{ji_1i_2k} = 1 \tag{4}$$

La restricción 4 garantiza que solo se pueda procesar un trabajo en una máquina a la vez.

$$C_{j_2ik} \geq C_{j_1ik} + p_{j_2ik} - P(3 - X_{j_1j_2i} - Y_{j_1k} - Y_{j_2k}), \forall k, i, j_1 < n, j_2 > j_1 \tag{5}$$

$$C_{j_1 i} \geq C_{j_2 i} + p_{j_1 i} - P(3 + X_{j_1 j_2 i} - Y_{j_2 k} - Y_{j_1 k}), \forall k, i, j_1 < n, j_2 > j_1 \quad (6)$$

Las restricciones 5 u 6 garantizan que cualquier máquina pueda procesar como máximo una operación a la vez. Se puede observar que la parte posterior de P es una condición de activación de la restricción. Si se anula, hace que la restricción sea trivial y que no afecte.

La ecuación 5 asegura que un trabajo no puede comenzar en una máquina hasta que el trabajo anterior no haya terminado. Evita solapamientos en la asignación de trabajos a máquinas dentro de una misma fábrica.

La ecuación 6 es similar a la anterior, solo que impone la relación inversa. Se usa para forzar un orden consistente entre los dos trabajos, evitando inconsistencias en la asignación de trabajos a máquinas. En resumen, aseguran que si j_2 va después de j_1 entonces j_2 comience después de que termine j_1.

$$C_{\text{máx}} \geq C_{jik}, \quad \forall j, i, k \quad (7)$$

La ecuación 7 calcula el *makespan* global. El *makespan* debe ser mayor o igual que el tiempo de finalización del trabajo que termine más tarde. Hay que recordar que el objetivo es minimizar el *makespan* 1.

Por último, se establecen las restricciones sobre las variables de decisión.

$$C_{jik} \geq 0, \quad \forall j, i, k \quad (8)$$

La ecuación 8 representa el tiempo de finalización del trabajo j en la máquina i en la fábrica k. Se define como una variable no negativa ya que no tiene sentido tiempos negativos.

$$Y_{jk} \in \{0, 1\}, \quad \forall j, k \quad (9)$$

La ecuación 9 define Y_{jk} como una variable binaria, permite que a cada trabajo se le asigne exactamente a una fábrica.

$$X_{j_1 j_2 i k} \in \{0, 1\}, \quad \forall j_2 < n, j_1 > j_2, i, k \quad (10)$$

La ecuación 10 define X como una variable binaria, como sus respectivos subíndices.

$$C_{\text{máx}} \geq 0 \quad (11)$$

Esta última ecuación 11 define el *makespan* como una variable no negativa, ya que no tiene sentido tener valores negativos.

Se puede observar cómo la formulación del problema es compleja, teniendo en cuenta que la cantidad de restricciones y de subíndices hace que el seguimiento y comprensión del problema puedan ser arduos. Para comprenderlo mejor, se explica un ejemplo.

3.2. Ejemplo Distributed Job-Shop Scheduling Problem

A continuación se mostrará un ejemplo del problema *Distributed Job-Shop Scheduling Problem* (DJSSP) con 4 trabajos, 2 máquinas y 2 fábricas para entender mejor el problema planteado.

Un aspecto importante es entender el formato de los datos. En primer lugar, aparecen las máquinas, fábricas y los trabajos; a continuación, se tiene una tabla.

(1,10)	(2,15)	(1,20)	(2,20)
(1,5)	(2,10)	(2,5)	(1,10)
(2,10)	(1,15)	(2,30)	(1,15)
(1,15)	(2,5)	(1,10)	(2,5)

Tabla 1: Datos ejemplo 1

Las filas de la Tabla 1 corresponden a los trabajos; como hay 4 trabajos, hay 4 filas. En cuanto a las columnas, habrá $m * f$. Las m primeras corresponden al orden de procesamiento de ese trabajo en cada máquina y su tiempo en la primera fábrica, y las siguientes columnas ocurre lo mismo, pero si entrase en diferentes fábricas. Es decir, si $J1$ fuese a ser procesado en $F1$, deberá entrar primero en $M1$, que tardará 10, y luego entrar en $M2$, que tardará 15. Si entrase en $F2$, el orden de procesamiento sería el mismo, pero el tiempo aumenta. Hay que tener en cuenta que el orden de procesamiento también puede ser modificado al estar en fábricas diferentes, como ocurre con $J2$.

Si resolvemos este ejemplo, obtenemos como solución los siguientes gráficos:

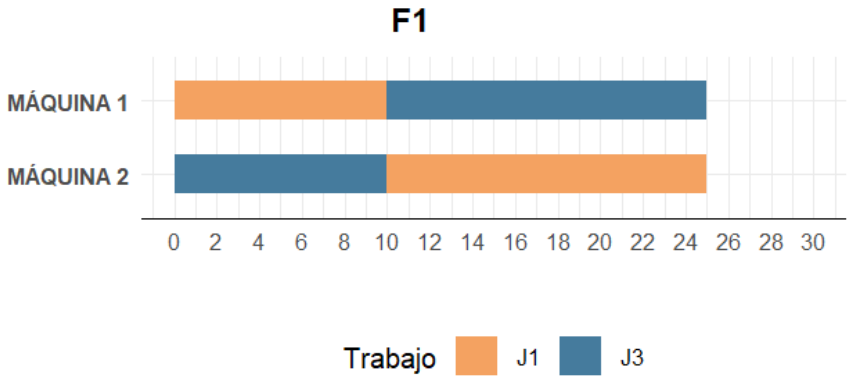

Figura 1: Distribución óptima $F1$ ejemplo 1

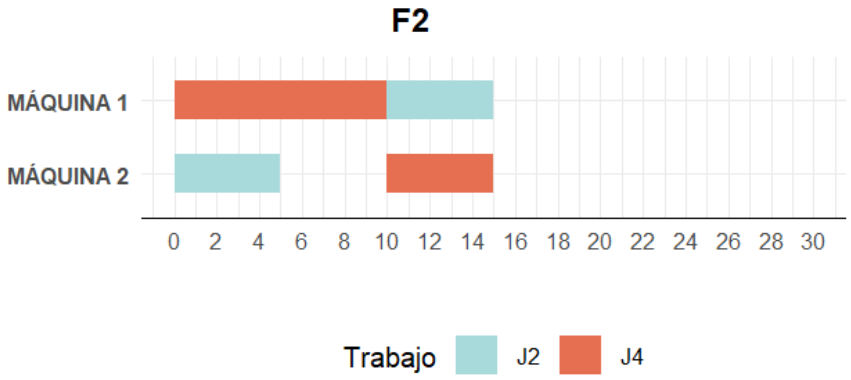

Figura 2: Distribución óptima $F2$ ejemplo 1

En la Figura 1 se observa la distribución óptima de los trabajos en la fábrica 1. Debido a ser un ejemplo bastante simple, se observa a simple vista cómo los dos trabajos utilizan la totalidad del tiempo disponible, adaptándose perfectamente entre sí.

En la Figura 2 se observa también la distribución óptima de los tiempos de los trabajos en la fábrica 2. En este caso, puede resultar más difícil de interpretar, ya que el trabajo dos genera un 'hueco' al tener que esperar entre sus procesos en las dos máquinas.

En este ejemplo, se llega a la conclusión de que el tiempo óptimo es 25. Debido a la sencillez del ejemplo, se puede ir analizando uno por uno los movimientos hasta llegar a la solución óptima.

En primer lugar, hay que darse cuenta de que si el $J1$ y $J3$ entrasen en $F2$, sería menos óptimo, ya que el tiempo de procesamiento es mayor, por lo que se obliga a que entre en $F1$. $J2$, tarda lo mismo independientemente de la fábrica, así que se puede dejar para el final y meterlo donde mejor encaje. Lo mejor para $J4$ sería

procesarse en la $F2$.

Una vez los trabajos estén distribuidos por fábricas, se mira el orden de procesamiento. Si observamos la Figura 1, en $F1$, da igual el orden de procesamiento de $J1$ y $J3$, ya que, mientras uno está en una máquina, el otro está en la otra, y no se solapan. Sin embargo, el problema estaría en $F2$, en la Figura 2 se puede observar cómo se procesa primero $J4$; sin embargo, si se procesara primero $J2$ el *makespan* aumenta.

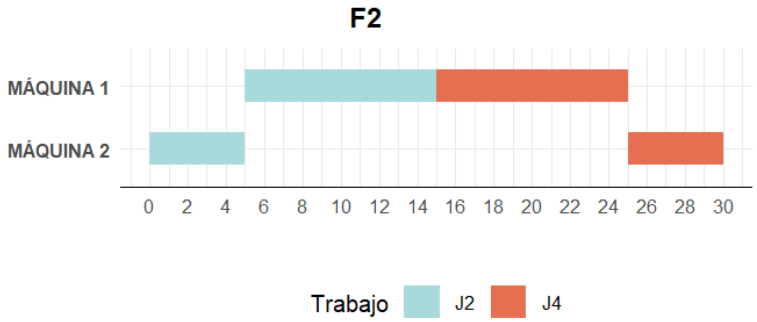

Figura 3: Orden incorrecto ejemplo 1

Analizando la Figura 3, se puede ver cómo simplemente por haber cambiado de orden dos trabajos en $F2$ hemos pasado de una solución de 25 a 30, lo que supone un aumento considerable del 20 %, lo que podría conllevar un aumento de los costes, una producción más ineficiente, entre otros problemas.

Gracias a este ejemplo se puede observar la importancia de seleccionar de forma correcta el orden de procesamiento y a qué fábrica se asigna cada trabajo. Por esa razón se van a proponer diferentes algoritmos que resuelvan el *DJSSP* para cualquier conjunto de datos en los que no se pueda ir analizando trabajo por trabajo como en el ejemplo.

4. Revisión bibliográfica

En esta sección se repasa toda la bibliografía utilizada para entender el problema y su contexto.

Los *Scheduling Problems*, conocidos en español como problemas de planificación de la producción, se centran en la asignación de recursos y la secuenciación de actividades para cumplir con objetivos como minimizar el tiempo total de ejecución o maximizar el uso de los recursos ...

El concepto de planificación de la producción tiene su origen en la década de 1950, cuando se empieza a planificar proyectos de este estilo. La introducción del Método de la Ruta Crítica (Critical Path Method, CPM) por Kelley y Walker en 1959 permitió identificar las actividades críticas que determinan la duración de un proyecto; este es considerado el primer problema de asignación de tareas, que supuso una gran revolución en la gestión eficiente de proyectos [2].

A partir de esa fecha surgen diversos tipos de problemas relacionados con la asignación de tareas, entre los que destacan los siguientes:

- **Programación de máquinas paralelas:** Este tipo de problemas se centran en asignar trabajos a varias máquinas que operan en paralelo. Las primeras heurísticas las propusieron Kurz y Askin en 2001 [7]. A partir de entonces surgen nuevos algoritmos y restricciones sobre este problema [8].

- **Programación de talleres de flujo:** También conocido como *Flow Shop Scheduling*, es similar al anterior pero todos los trabajos siguen la misma secuencia de máquinas.

- Existen muchos otros problemas pero no es el objetivo de este estudio. Solo se explicará el problema principal *Job Shop Scheduling* y su versión *Distributed*.

Uno de los principales problemas del artículo de Kelley y Walker es que no abordaban la secuenciación de trabajos en entornos con múltiples máquinas y rutas variables. Por esa razón surge en *Job Shop Scheduling Problem*.

Una de las características de *JSSP* es que es un problema NP-Hard; se han desarrollado diversas estrategias para abordar la obtención de una solución al menos aproximada.

En 1964, Roy y Sussman proponen una primera representación mediante grafos del *JSSP* [9]. Pero no sería hasta en 1969 donde Balas usaría esta representación para resolver el problema empleando el concepto de camino crítico [10].

Los primeros métodos exactos aportaban un gran valor teórico, pero, como ya se ha dicho antes, al ser un problema NP-Hard, estos métodos solo podían resolver problemas reducidos, con pocos trabajos y pocas máquinas.

Debido a la justificación de la complejidad del problema, se empezaron a proponer diferentes heurísticas, sobre todo algoritmos genéticos [5]. También, se empezaron a usar en los años 80 diferentes redes neuronales para poder resolver este complejo problema.

El *Distributed Job Shop Scheduling Problem* surge como una extensión del *JSSP* para considerar la coordinación de múltiples fábricas, encontrándose así diferentes desafíos y problemas a la hora de adaptar los algoritmos.

Actualmente existen pocos algoritmos propuestos para resolver el *DJSSP*. Algunos de los más interesantes se pueden encontrar en el *paper* [6], donde se propone un algoritmo *HGTSA* (Hybrid Genetic Tabu Search Algorithm), un algoritmo híbrido que combina el algoritmo genético y la búsqueda tabú. Sin embargo, en ese mismo *paper* se encuentran varios errores en la formulación matemática, como ya se ha explicado en la sección anterior. Además, para la resolución del problema solo se plantean fábricas homogéneas.

El problema *DJSSP* con fábricas heterogéneas es un problema actualmente tratado y sobre el que no hay muchas referencias al respecto.

5. Técnicas de análisis comparativo de algoritmos de solución al problema DJSSP

Una vez entendido el problema, se proponen diferentes algoritmos para resolver este problema, llegando a una solución lo más cercana a la óptima posible.

Sin embargo, una parte muy importante es la comparación de esos algoritmos para ver si hay diferencias significativas de uno a otro.

Para compararlos, primero hay que entender qué resultados se van a obtener.

	Algoritmo 1	Algoritmo 2	Algoritmo 3
Datos 1	319	350	290
Datos 2	540	530	530
Datos 3	420	435	419

Tabla 2: Soluciones ejemplo

En la Tabla 2 se puede ver cómo las columnas son los algoritmos y las filas son los datos. Los datos son conjuntos de prueba los cuales tienen unos trabajos, máquinas y fábricas que sirven para simular y probar los diferentes algoritmos. La Tabla 2 simplemente muestra cómo es el formato obtenido al probar los diferentes algoritmos en los diferentes datos; es una tabla de ejemplo. Los números que hay son el *makespan* obtenido al probar los algoritmos en los diferentes conjuntos de datos, siendo esto lo que interesa comparar, aunque hay que tener en cuenta los tiempos de ejecución también.

Destacar que estos tiempos y *makespan* no son directamente comparables con diferentes conjuntos de datos. Esto es debido a que no solo se van a comparar conjuntos de datos de distintos tamaños, sino que también conjuntos de datos del mismo tamaño podrían contener tareas más complejas o menos, lo que daría como resultado diferentes tiempos y *makespan* que no serían comparables entre sí.

Para comparar los resultados obtenidos en los diferentes algoritmos se utilizarán los siguientes tests.

5.1. Test de Friedman

El test de Friedman [11] es una prueba no paramétrica desarrollada por Milton Friedman; proviene de una extensión de la prueba de suma de rangos con signo de Wilcoxon. Se utiliza para detectar diferencias en tratamientos múltiples (en este caso, algoritmos) cuando las observaciones están organizadas en bloques (en este caso, conjuntos de datos).

La prueba consiste en ordenar los resultados dentro de cada bloque. Para cada conjunto de datos i, los k algoritmos son ordenados del mejor (1) al peor (k). Esto se denomina *Rankings* y este orden generado sí que es comparable entre diferentes conjuntos de datos.

El estadístico de Friedman se calcula como:

$$\chi_F^2 = \frac{12N}{k(k+1)} \left[\sum_{j=1}^{k} R_j^2 - \frac{k(k+1)^2}{4} \right] \tag{12}$$

Siendo N el número de conjuntos de datos, k el número de algoritmos que se comparan, y R_j la suma de rangos para el tratamiento j.

Bajo la hipótesis nula, el estadístico sigue una distribución X^2 con $k-1$ grados de libertad [12].

La hipótesis nula H_0 establece que todos los algoritmos tienen un rendimiento equivalente, mientras que la hipótesis alternativa H_1 indica que al menos un algoritmo difiere significativamente.

El resultado de la prueba indica si hay diferencias entre los diferentes algoritmos a comparar. Posteriormente, habría que determinar cuáles son diferentes. Para ello, se podrían usar comparaciones Post-Hoc como el test de Tukey o el test de rangos con signo de Wilcoxon.

Este test ha sido bastante criticado debido a ser un test bastante exigente con ciertas limitaciones, entre las que destacan.

- Tiende a ser menos potente que alternativas paramétricas como ANOVA.
- Requiere ajustes en el caso de que haya empates.
- Asume que las diferencias entre pares están igualmente distribuidas.

Cuando el test de Friedman encuentra diferencias significativas, se recomienda hacer pruebas Post-Hoc de Nemenyi para comparaciones múltiples.

Para implementar este test en R no es necesario instalar ninguna librería adicional, ya que pertenece a la librería *stats* que viene por defecto en la instalación de R.

```
friedman_makespan <- friedman.test(
        Makespan ~ Algoritmo | Datos,
        data = datos_makespan_melt
        )
print("Resultado del test de Friedman (Makespan):")
print(friedman_makespan)
```

Para el tests se necesita que los datos estén en formato largo, es decir, las columnas son Datos, Algoritmos y la media a analizar (Tiempo o *makespan*). El algoritmo es el grupo (los diferentes algoritmos que se comparan) y los Datos son los diferentes conjuntos de datos. El resultado del test indica si existen o no diferencias significativas entre al menos dos de los algoritmos comparados.

5.2. ANOVA de medidas repetidas

El ANOVA de medidas repetidas es la alternativa paramétrica al test de Friedman cuando se cumplen sus supuestos estadísticos [13].

El diseño es apropiado porque:

- Cada algoritmo se aplica a los mismos conjuntos de datos (medidas repetidas).
- Aísla la variabilidad entre conjuntos de datos (efecto bloque).
- Mayor potencia que alternativas no paramétricas cuando se cumplen los supuestos.

Mientras que Friedman usa los rangos, el ANOVA de medidas repetidas opera sobre los valores originales, ofreciendo:

- Mayor potencia para detectar diferencias pequeñas.
- Capacidad de cuantificar el tamaño del efecto.
- Posibilidad de modelar interacciones.

Para su validez se requiere:

- Varianzas iguales entre niveles.
- Residuales normalmente distribuidos (QQ-plot o Shapiro-Wilk)
- Datos en escala métrica.

Si estos supuestos fallan, es preferible recurrir solo al test de Friedman, aunque sea más exigente.

La hipótesis nula H_0 establece la igualdad en las medias poblacionales: $H_0 : \mu_1 = \mu_2 = ... = \mu_k$

La hipótesis alternativa H_1 indica diferencias significativas.

Al igual que ocurría en el test de Friedman, si se rechaza H_0 se deberían aplicar comparaciones post-hoc para identificar las diferencias específicas.

El ANOVA de medidas repetidas tiene ventajas sobre el de Friedman, aunque también tiene limitaciones. Por esa razón, se intentará usar los dos para complementar la comparación en los casos que sea posible.

Para implementar este test en R es necesario tener los datos en un formato de tabla, siendo cada fila un sujeto (conjunto de datos) y cada columna un algoritmo.

Además, son necesarias las siguientes librerías.

- *stats*: Librería base de R para el ANOVA.
- *car* o *afex*: Librerías para comprobar la esfericidad.
- *ggpubr*: Librería para el gráfico QQ de normalidad.
- *car*: Librería para comprobar la homogeneidad de varianzas.

```
modelo_aov <- aov(Makespan ~ Algoritmo + Error(Datos/
   Algoritmo),
                  data = datos_makespan_melt)
summary(modelo_aov)
```

Con esta función se implementa en R un ANOVA de medidas repetidas, donde se introduce una variable dependiente (*makespan*), el algoritmo y los casos. Además, se introduce una estructura de error para las medidas repetidas.

Además, hay que verificar ciertos supuestos de la siguiente forma.

```
# 1. Esfericidad (Mauchly)
check_sphericity(modelo_afex)

# 2. Normalidad (QQ-plot y Shapiro)
residuos <- residuals(modelo_afex)
ggqqplot(residuos) + stat_qqline()
shapiro.test(residuos)

# 3. Homogeneidad de varianzas (Levene)
car::leveneTest(Makespan ~ Algoritmo,
        data = datos_makespan_melt)
```

Con este código se verifica si cumple los supuestos de esfericidad, normalidad y homogeneidad de las varianzas.

5.3. Test de Nemenyi

El test de Nemenyi [14] es una prueba no paramétrica de comparaciones múltiples derivada de la metodología de rango studentizado de Tukey. Este test implementa tras un test de Friedman o de ANOVA de medidas repetidas significativo ($p < 0{,}05$) para poder identificar las diferencias específicas entre pares de algoritmos.

Con este test se controla el error de Tipo I en comparaciones múltiples mediante la aproximación de rango studentizado:

$$CD = q_{\alpha,k,\infty}\sqrt{\frac{k(k+1)}{12N}} \tag{13}$$

Donde CD es la diferencia crítica, q el valor crítico de la distribución, k el número de algoritmos y N el número de bloques (conjunto de datos).

Su implementación en R es de la siguiente forma:

```
nemenyi_makespan <- PMCMRplus::frdAllPairsNemenyiTest(
  Makespan ~ Algoritmo | Datos,
  data = datos_makespan_melt
)
```

Se necesita el paquete *PMCMRplus*, que es una librería especializada en pruebas post-hoc no paramétricas para comparaciones múltiples.

En la salida se podrá observar una matriz de p-valores que indican las diferencias significativas entre pares, además de los estadísticos de diferencia y la diferencia crítica calculada.

La combinación de los dos tests explicados anteriormente y del test de Nemenyi ayuda a analizar si existen diferencias entre los distintos algoritmos que se prueban.

6. Instancias utilizadas

El estudio emplea 40 instancias del problema *DJSSP*, divididas en dos categorías principales según el tamaño del problema.

- Instancias pequeñas: compuestas por 20 Trabajos, 3 Máquinas y 2 Fábricas.
- Instancias pequeñas: compuestas por 50 Trabajos, 3 Máquinas y 3 Fábricas.

Cada instancia se representa como una lista de trabajos, donde cada trabajo contiene pares (máquina, tiempo) para sus operaciones en las diferentes fábricas. Estas listas varían en función de la fábrica, reflejando la naturaleza heterogénea del entorno distribuido.

El formato de representación utilizado sigue el mismo esquema detallado en el 3.2, donde se explica cómo se codifican los trabajos y las operaciones en cada fábrica y en cada máquina.

La principal razón por la que se dividen las instancias en esas dos categorías es la diversidad computacional. Las instancias pequeñas (20 trabajos) permiten evaluar la eficiencia en tiempos reducidos. Las instancias grandes (50 trabajos) prueban la escalabilidad y robustez del algoritmo ante espacios de búsqueda complejos, ya que existen muchas más posibles soluciones y combinaciones diferentes de trabajos.

Por otro lado, en las diferentes instancias destaca la heterogeneidad de tiempos, ya que contienen tiempos de procesamiento de los trabajos muy dispares, de tal forma que se puedan probar los algoritmos en diferentes situaciones.

Además, las instancias fueron generadas con un cierto grado de aleatoriedad controlada, es decir, aunque se incluyen elementos aleatorios, se aplican criterios para evitar que los datos generen patrones predecibles. Esto es esencial para evitar el sobreajuste, es decir, que un algoritmo funcione bien únicamente en un tipo de instancia con ciertas regularidades artificiales, pero falle en contextos reales o más amplios.

7. Algoritmos para la solución del DJSSP

Una vez ya sabemos a qué problema hay que enfrentarse, y cómo se compararán los resultados, sabiendo que el objetivo de estos algoritmos es minimizar el tiempo en el que se completan todos los trabajos (*makespan*), se plantean diferentes algoritmos con el objetivo de encontrar soluciones aproximadas a la solución óptima.

Para obtener más información sobre la programación de los algoritmos que se explican a continuación, consultar el Anexo 9.

7.1. Heurística greedy

La heurística greedy, también conocida como algoritmos voraces, es una estrategia con la que se pretende obtener una solución factible (puede ser óptima o no). La forma de llegar a esa solución es elegir soluciones locales de forma óptima, o que cumplan ciertos criterios [15].

Aplicado al *DJSSP*, se irán seleccionando trabajos secuencialmente en base a un criterio específico y se introducirán en la solución. Para comparar los resultados de los criterios se usará tanto el *makespan* obtenido, como el tiempo.

Se va a probar con 3 criterios greedy diferentes para poder comparar los resultados. Los criterios son:

- Criterio 1: Seleccionar el trabajo que menos tiempo necesite para completarse. Se recorrerá cada trabajo sumando lo que se tarda en cada máquina. De esta forma se obtiene el trabajo y la fábrica que menos tarda de todos. La ventaja de este criterio es que se reduce el tiempo de ocupación de recursos, pero la limitación es que ignora el impacto en el *makespan* global.

- Criterio 2: Seleccionar el trabajo que minimice el *makespan*. Se obtiene una copia de la solución anterior, y se simula el *makespan* si se introdujese cada trabajo en cada fábrica. De esta forma seleccionamos el trabajo y la fábrica que minimice el *makespan*. La ventaja es que es un enfoque muy directo en la optimización global, pero tendrán un mayor costo computacional por simulaciones.

- Criterio 3: Seleccionar el trabajo con la penalización mayor. Para cada trabajo se busca la fábrica que tarde más y la que tarde menos. La penalización es la diferencia de esos tiempos. Se introduce en la solución el que mayor penalización tenga. Una vez tenemos el trabajo al igual que en el criterio 2, se prueba que fábrica minimiza el *makespan* para el trabajo obtenido, ya que podría haber más de 2 fábricas e introducir el trabajo en la fábrica que menos

tarda no tendría porque ser lo más eficiente. La ventaja es que permite balancear la carga en sistemas heterogéneos, pero es muy sensible a distribuciones asimétricas de tiempos.

Para entender mejor el algoritmo Greedy, se puede observar en el siguiente pseudocódigo:

Algoritmo 1 Greedy

```
1: Inicializar los tiempos de inicio y finalización del trabajo en las máquinas
2: Inicializar el tiempo final
3: while Queden trabajos sin asignar do
4:     Para cada trabajo pendiente y cada fábrica:
5:         Se busca el trabajo y la fábrica que cumpla el criterio seleccionado.
6:     Seleccionar la combinación (trabajo, fábrica) que cumpla con el menor
       tiempo (Depende del criterio Greedy).
7:     Asignar ese trabajo a la fábrica seleccionada.
8:     Actualizar los tiempos de las máquinas correspondiente
9:     Se añade el mejor trabajo y fábrica a la solución
10: end while
11: return La solución construida.
```

La idea del Algoritmo 1 es clara, primero tener en cuenta que no deben quedar trabajos sin asignar. Después, para cada trabajo sin asignar se busca el que cumpla con el criterio Greedy seleccionado, y el que cumpla se introduce en la solución, se actualizan los tiempos correspondientes y se sigue con el procesamiento del resto de los trabajos.

Una vez se ha seleccionado un trabajo y su fábrica, hay que meterlo en la solución. Para ello se usa una función la cual recorre las máquinas necesarias para su procesamiento, buscando un espacio en el que se pueda procesar sin interrumpirse. Para lograrlo, se recorren los trabajos que ya están en la solución en esa máquina, y se busca el menor; se intenta introducir ahí si hay espacio suficiente, si no, se sigue buscando. Hay que tener en cuenta que la segunda máquina de procesamiento tiene que empezar una vez terminado de procesar el trabajo en la primera máquina, por lo que hay que ver que ese tiempo mínimo que se ha comentado antes sea mayor que el tiempo mínimo requerido para cumplir las especificaciones del problema.

Para entender mejor las diferencias entre los tres criterios Greedy propuestos, se realiza un análisis comparativo.

Criterio	Ventaja	Desventaja
Greedy C1	Rápido, bajo costo computacional	Subóptimo para *makespan* global
Greedy C2	Optimización global	Costoso para grandes instancias
Greedy C3	Efectivo en máquinas heterogéneas	Requiere cálculos adicionales

Tabla 3: Criterios Greedy

7.1.1. Comparación de resultados

Para comparar los tres criterios greedy se van a usar diferentes instancias para poder comparar tanto el *makespan* como el tiempo.

Las instancias corresponden a conjuntos específicos de trabajos, fábricas y tiempos de procesamiento, utilizados para evaluar el rendimiento del algoritmo en distintos escenarios.

En primer lugar, se va a comparar instancias de distintos tamaños para valorar, sobre todo, el tiempo de cómputo.

	Greedy C1		Greedy C2		Greedy C3	
Tamaño instancia	Makespan	Tiempo	Makespan	Tiempo	Makespan	Tiempo
8	397	0.0056	343	0.0041	352	0.0085
16	565	0.0051	527	0.0509	584	0.0167
50	565	0.0289	552	0.5558	610	0.0604
100	983	0.1282	952	8.2633	999	0.2730
200	1743	0.6759	1679	137.8194	1679	2.1122

Tabla 4: Análisis comparativo de heurísticas greedy de distintos tamaños

Analizando los tiempos de ejecución en la Tabla 4, se puede observar cómo, a partir de 100 trabajos, el tiempo aumenta considerablemente; se puede ver más claro gracias al siguiente gráfico.

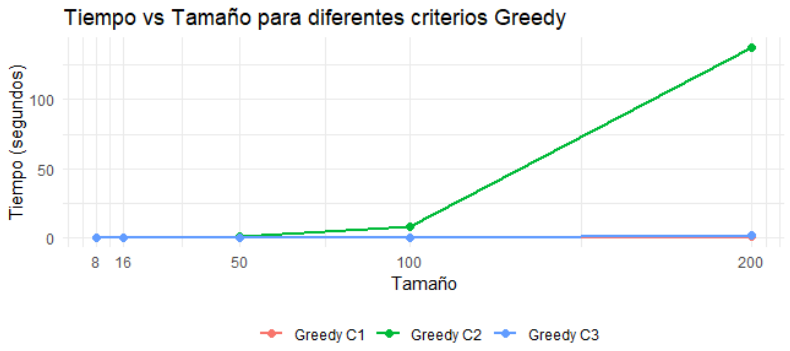

Figura 4: Gráfico: Tamaño vs Tiempo

Pese a que los criterios uno y tres mantienen un tiempo cercano a 0, se puede observar en la Figura 4 cómo el tiempo del criterio 2 aumenta considerablemente a partir del tamaño 100.

Estos tiempos son con el algoritmo Greedy; con otros algoritmos como el GRASP, los tiempos son muy superiores, lo que no facilita la comparación de algoritmos.

Es por esta razón por la que, para la comparación de algoritmos, se usarán instancias diferentes de tamaño 20 y 50.

Resultados con instancias de tamaño 20 y 50:

	Greedy C1		Greedy C2		Greedy C3	
Datos	Makespan	Tiempo	Makespan	Tiempo	Makespan	Tiempo
dj20-1	349	0.0112	372	0.0235	345	0.0508
dj20-2	402	0.0183	380	0.0207	338	0.0142
dj20-3	408	0.0254	320	0.0197	306	0.0150
dj20-4	345	0.0148	349	0.0215	340	0.0118
dj20-5	338	0.0163	344	0.0186	350	0.0157
dj20-6	348	0.0204	361	0.0186	355	0.0144
dj20-7	470	0.0126	302	0.0188	314	0.0114
dj20-8	352	0.0192	334	0.0213	313	0.0151
dj20-9	352	0.0172	328	0.0176	300	0.0124
dj20-10	418	0.0193	403	0.0175	374	0.0232
dj20-11	326	0.0183	318	0.0190	325	0.0171
dj20-12	363	0.0174	324	0.0181	276	0.0179
dj20-13	390	0.0142	381	0.0201	316	0.0139
dj20-14	364	0.0173	367	0.0223	301	0.0172
dj20-15	333	0.0180	324	0.0197	316	0.0151
dj20-16	331	0.0186	346	0.0188	332	0.0153
dj20-17	359	0.0181	285	0.0177	295	0.0167
dj20-18	342	0.0169	360	0.0195	330	0.0144
dj20-19	354	0.0183	371	0.0176	318	0.0123
dj20-20	370	0.0218	320	0.0166	323	0.0146
dj50-1	436	0.0261	446	0.3255	437	0.0484
dj50-2	542	0.0322	474	0.3532	514	0.0441
dj50-3	543	0.0276	450	0.3587	452	0.0445
dj50-4	496	0.0262	438	0.3794	433	0.0451
dj50-5	537	0.0247	473	0.3557	451	0.0511
dj50-6	492	0.0277	469	0.3697	465	0.0514

	Greedy C1		Greedy C2		Greedy C3	
Datos	Makespan	Tiempo	Makespan	Tiempo	Makespan	Tiempo
dj50-7	686	0.0242	477	0.3965	489	0.0496
dj50-8	525	0.0265	528	0.3457	470	0.0514
dj50-9	530	0.0308	446	0.4084	467	0.0465
dj50-10	479	0.0281	461	0.3500	446	0.0446
dj50-11	564	0.0304	469	0.3665	477	0.0469
dj50-12	436	0.0258	429	0.3957	417	0.0460
dj50-13	562	0.0295	464	0.3637	477	0.0464
dj50-14	529	0.0362	459	0.3568	460	0.0649
dj50-15	529	0.0377	462	0.3637	463	0.0434
dj50-16	418	0.0264	366	0.4082	386	0.0587
dj50-17	604	0.0277	462	0.3832	473	0.0445
dj50-18	569	0.0186	479	0.4222	414	0.0498
dj50-19	639	0.0243	440	0.4046	439	0.0619
dj50-20	469	0.0317	411	0.4550	381	0.0455

Tabla 5: Resultados greedy. 20 y 50 trabajos

Una vez obtenidos los resultados que se pueden observar en la Tabla 5, se procede a su comparación.

En primer lugar, se comparará el tiempo de los algoritmos, para tenerlo en cuenta posteriormente al comparar el *makespan*.

Recordar lo explicado en 5.1 sobre los rankings.

Se obtienen los rankings para el tiempo:

Algoritmo	**Ranking Tiempo**
GreedyC1	1.58
GreedyC2	2.8
GreedyC3	1.62

Tabla 6: Rankings Greedy Tiempo

Observando la Tabla 6 se observa cómo el criterio uno es el ganador (el que suele ir más rápido), pero seguido muy de cerca por el criterio tres.

En primer lugar, se aplica el test de Friedman, obteniendo como resultado un estadístico de 38.45 y un p-valor de 4.47e-09. Como el p-valor es menor que 0.05,

se rechaza la hipótesis nula; por lo tanto, sí existen diferencias significativas entre los algoritmos en tiempo de ejecución.

Con el ANOVA de medidas repetidas se confirma lo expuesto anteriormente, con un p-valor muy pequeño de 1.64e-12, se puede decir que hay diferencias significativas entre los tiempos de los tres criterios greedy. Sin embargo, se están violando los supuestos del ANOVA explicado anteriormente. Las diferencias detectadas son reales, pero los p-valores del ANOVA podrían estar subestimando.

Para ver entre qué algoritmos existen diferencias, se usa el test de Nemenyi, obteniendo los siguientes resultados por pares.

- GreedyC1 vs. GreedyC2: p-valor = 1.3e-07.
- GreedyC1 vs. GreedyC3: p-valor = 0.97.
- GreedyC2 vs. GreedyC3: p-valor = 4.4e-07.

Se puede observar cómo existen diferencias significativas entre el tiempo de ejecución del GreedyC1 y del GreedyC2; también hay diferencias entre GreedyC2 y GreedyC3. Por lo tanto, se podría decir que el tiempo de ejecución del criterio greedy C2 es significativamente diferente de los otros criterios greedy.

Una vez se saben las diferencias de los criterios en cuanto a tiempo de ejecución, se procede a comparar el *makespan*.

Se obtienen los rankings para el *makespan*:

Algoritmo	**Ranking**
GreedyC1	2.65
GreedyC2	1.88
GreedyC3	1.48

Tabla 7: Rankings Greedy

En la Tabla 7 se observa cómo el criterio tres es el ganador, estando bastante lejos el criterio uno.

Aplicando de nuevo el test de Friedman, pero esta vez para comparar el *makespan*, se obtiene un p-valor de 6.316e-07. Por lo que se rechaza la hipótesis nula, es decir, sí que hay diferencias entre los diferentes criterios greedy.

Con el test de Nemenyi se realizan las comparaciones por pares, obteniendo:

- GreedyC1 vs. GreedyC2: p-valor = 0.0015.

- GreedyC1 vs. GreedyC3: p-valor = 4.4e-07.
- GreedyC2 vs. GreedyC3: p-valor = 0.1733.

Por lo tanto, sí que existen diferencias significativas ($\alpha = 0{,}05$) entre el Greedy C1 y Greedy C2, y entre Greedy C1 y Greedy C3. Sin embargo, no hay diferencias entre Greedy C2 y Greedy C3.

Con el ANOVA se obtiene un p-valor de 8.35e-12, por lo que las diferencias del *makespan* entre los diferentes criterios sí que son significativamente diferentes. Sin embargo, ocurre igual que en el caso anterior: se está violando la normalidad y la homogeneidad de varianzas.

Como conclusión final sobre los criterios Greedy, el criterio C2 es el que más tarda con mucha diferencia, y es significativamente diferente de los otros tiempos del resto de los criterios. Esta conclusión es lógica ya que para decidir cuál entra en la solución se prueba trabajo por trabajo para minimizar el *makespan* por lo que el criterio C2 es $O(n^2)$. Sin embargo, en el *makespan* no hay diferencias significativas entre el criterio C2 y C3, siendo el criterio C1 el que peores resultados ofrece.

Por lo tanto, aunque lo mejor sería probar siempre todos los criterios posibles y ver cuál se adapta mejor a ese caso particular de la solución, se podría decir que el criterio C3 es el mejor en cuanto a la relación *makespan* tiempo. El criterio C2 da también buenos resultados, pero tarda bastante en ejecutarse, por lo que, dependiendo de la situación, puede no servir.

7.2. Metaheurísticas *GRASP*

GRASP es un algoritmo de optimización metaheurístico que combina los principios de algoritmos aleatorizados y técnicas de búsqueda local [16] [17]. Significa *"Greedy Randomized Adaptative Search Procedure"*, este procedimiento es iterativo, en el que con cada paso se intenta llegar a una solución más cercana a la óptima [18].

La eficacia en problemas Scheduling está demostrada en [19], especialmente para problemas como muchas instancias.

Como se ha comentado anteriormente, las metaheurísticas *GRASP* constan de varias fases claramente diferenciadas [18].

- Fase constructiva.
- Fase de mejora.
- Fase de actualización.

Algoritmo 2 Greedy Aleatorio (FASE CONSTRUCTIVA)

```
1: Inicializar las variables
2: for cada Trabajo do
3:     Se buscan los K trabajos con mejor criterio Greedy
4:     Se añaden a la lista restringida de candidatos
5:     Se elige un candidato aleatorio de la lista
6:     Ese trabajo y fábrica entran en la solución
7: end for
8: return solución
```

El primer paso es la "Fase constructiva"que se observa en el Algoritmo 2. Consiste en generar una lista restringida de candidatos mediante el uso de un algoritmo greedy; una vez se tiene la lista restringida de los mejores candidatos posibles, se selecciona un elemento de la lista de manera aleatoria, y ese es el elemento seleccionado que se introducirá en la solución. Se repite este mismo proceso hasta obtener la solución completa [18]. Este proceso es conocido como *Greedy Aleatorizado* y se usarán tres criterios greedy diferentes, los mismos que se explicaron en la sección Greedy.

Algoritmo 3 Búsqueda Local (FASE DE MEJORA)

```
1: Entrada: Una solución
2: Inicializar las variables
3: for cada Máquina do
4:     Se intercambian trabajos y se comprueba si mejora o no
5:     Si la solución mejora se guarda el resultado
6: end for
7: return solución mejorada
```

La segunda parte es la "Fase de mejora", que se observa en el Algoritmo 3, la cual consiste básicamente en una búsqueda local. Una vez obtenida una solución completa tras la fase constructiva, se buscará en el espacio de soluciones cercano una solución mejor. En el caso particular del problema, la búsqueda local que se propone es el intercambio de orden entre trabajos cercanos en una misma fábrica. Hay que tener cuidado con las búsquedas locales que se proponen, ya que en algunos casos complejos se podría llegar a comprobar todo el espacio de soluciones posibles, lo que equivale a un algoritmo de fuerza bruta, lo cual generaría muchos problemas

de tiempo de cómputo. Por esa razón, la búsqueda local propuesta solo intercambia los trabajos cercanos, para evitar aumentar el tiempo de cómputo.

Algoritmo 4 GRASP (FASE DE ACTUALIZACIÓN)

```
1: while Número máximo de iteraciones do
2:     SOLUCIÓN ← GreedyAleatorio()
3:     SOLUCIÓN ← BusquedaLocal(SOLUCION)
4:     Se guarda la mejor solución obtenida
5: end while
```

La tercera fase que se ha denominado "Fase de actualización", se observa en el Algoritmo 4. Consiste simplemente en valorar si la solución obtenida con la fase anterior es la mejor solución obtenida hasta el momento; en ese caso, se almacena la solución y se repite este proceso un número N de veces.

Para lograr unos buenos resultados, es necesario que el número de iteraciones (N) sea un número grande; en un principio, se propone que se realicen mil iteraciones.

Primero se aplican los algoritmos a instancias de distintos tamaños para poder tomar decisiones sobre el tiempo de ejecución.

	GRASP C1		GRASP C2		GRASP C3	
Tamaño	Makespan	Tiempo	Makespan	Tiempo	Makespan	Tiempo
8	264	1.8652	264	2.4394	269	1.6361
16	425	19.886	428	26.745	431	15.983
50	501	873.52	503	2404.7	480	847.87

Tabla 8: Resultados GRASP. Diferentes tamaños

Todos los resultados de la Tabla 8 han sido obtenidos con mil iteraciones. Se puede observar el aumento de tiempo considerable al aumentar el número de instancias, lo que hace inviable probar las metaheurísticas GRASP con conjuntos de datos que contengan más de cien o doscientos trabajos.

Además, para reducir el tiempo de cómputo, se analiza el *makespan* obtenido en cada iteración.

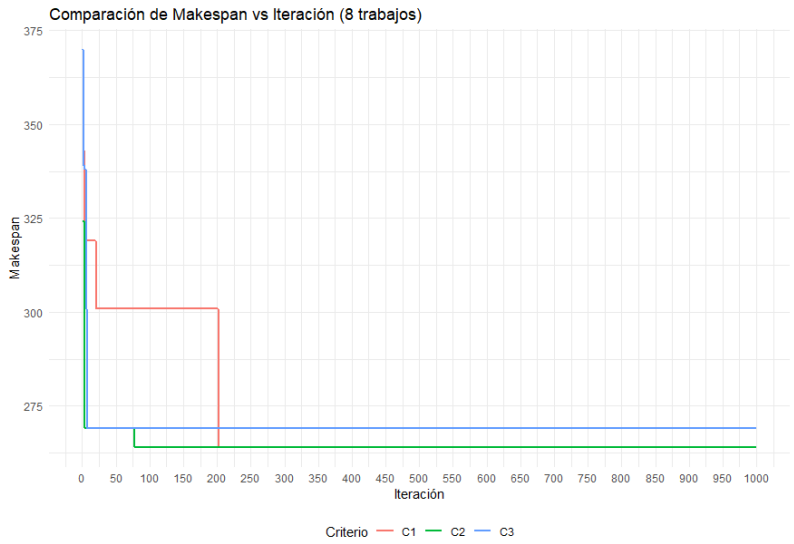

Figura 5: Comparación de Makespan vs Iteración (8 trabajos)

En esta primera Figura 5 se observa cómo a partir de doscientas iteraciones ya se llega a una solución estable, destacando cómo con el criterio dos y tres, en la iteración setenta y cinco, ya se había llegado a una solución estable.

Una solución es estable si, tras un cierto número de iteraciones, la solución no mejora o lo hace de forma insignificante.

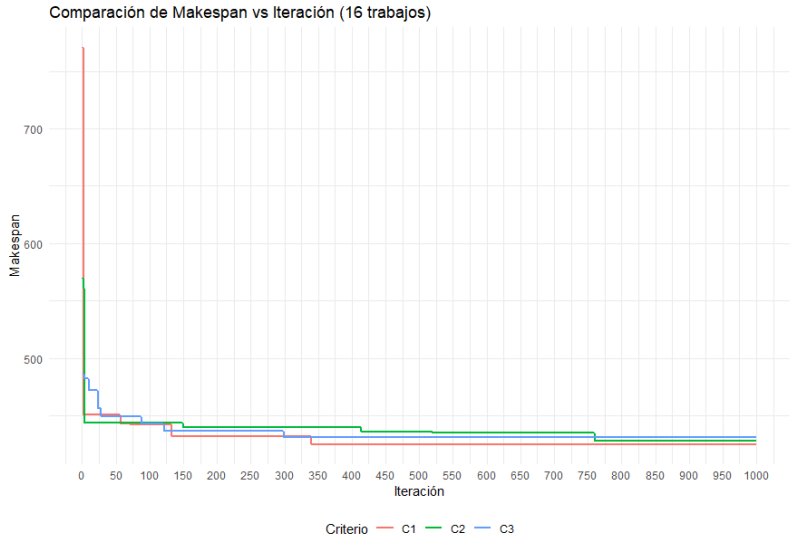

Figura 6: Comparación de Makespan vs Iteración (16 trabajos)

Observando la Figura 6 es interesante ver cómo son necesarias más iteraciones que en el caso anterior (8 trabajos) para llegar a la solución final.

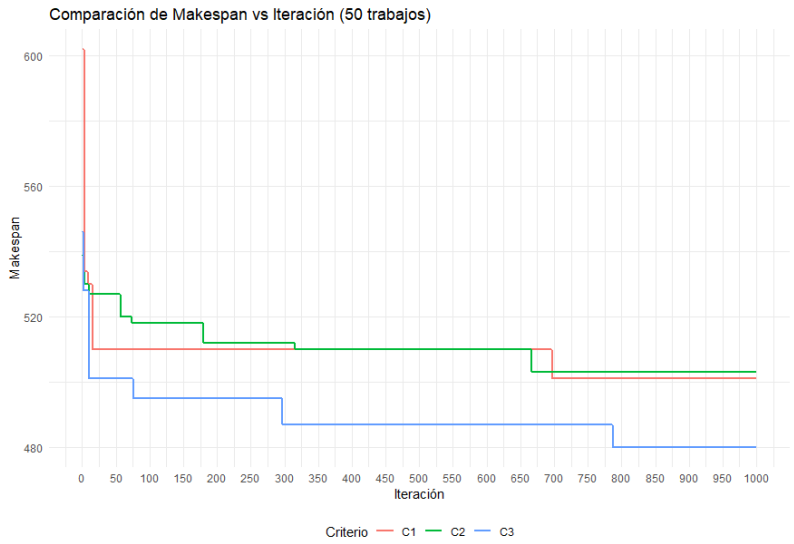

Figura 7: Comparación de Makespan vs Iteración (50 trabajos)

En esta última Figura 7 se ve cómo no se llega a una solución estable, por lo que cuantas más iteraciones haya, habrá una mejor solución.

De los tres gráficos anteriores se pueden obtener varias conclusiones. En las primeras cincuenta iteraciones hay una mejora bastante grande del *makespan*, por lo que es indispensable dejar el algoritmo un mínimo de setenta y cinco iteraciones. Además, se puede observar cómo cuantas más instancias (trabajos) hay, son necesarias un mayor número de iteraciones para obtener resultados cada vez mejores.

Sin embargo, para la comparación de algoritmos, que es lo que interesa, hay que intentar bajar el tiempo de cómputo para agilizar el proceso; por esa razón, se pondrá un límite de cien iteraciones sin actualizar, es decir, si se realizan cien iteraciones en las que no se obtiene una mejora de la solución final, se finaliza el bucle y se termina el algoritmo.

Es muy importante esta optimización de parámetros realizada, ya que es vital ajustar las iteraciones para ajustarse a los tiempos disponibles.

Es por esa razón, por la que aunque se usen los tiempos para hacerse una idea de lo que tarda en ejecutarse cada algoritmo, no se pueden comparar, ya que esos tiempos se obtienen en condiciones diferentes. Sin embargo, esto no es un problema, ya que como se ha visto anteriormente, la comparación de los tiempos de los criterios greedy ya se ha realizado, y en este caso se obtendrían las mismas conclusiones respecto al tiempo.

Resultados con instancias de tamaño 20 y 50:

	GRASP C1		GRASP C2		GRASP C3	
Datos	Makespan	Tiempo	Makespan	Tiempo	Makespan	Tiempo
dj20-1	291	11.195	287	6.4883	276	10.651
dj20-2	296	4.7710	288	10.115	282	8.5672
dj20-3	282	9.0962	267	11.065	267	5.4546
dj20-4	284	4.3722	284	7.0843	280	4.7779
dj20-5	289	6.2045	274	13.168	278	12.217
dj20-6	281	6.4606	277	7.3422	272	4.0075
dj20-7	267	10.268	266	16.788	258	3.4921
dj20-8	267	5.2371	265	6.2448	269	6.8307
dj20-9	258	7.3426	268	7.4576	264	6.6046
dj20-10	313	4.0513	317	8.1545	302	5.1501
dj20-11	273	5.5297	256	15.356	254	4.4487
dj20-12	245	5.5735	250	12.933	229	5.5775
dj20-13	291	5.7018	271	11.884	262	8.3780
dj20-14	277	4.4194	269	8.7713	261	9.4861
dj20-15	266	3.9803	259	6.8754	249	5.9235
dj20-16	286	5.4080	281	11.503	265	4.7815
dj20-17	259	6.9437	263	7.4917	250	4.4306
dj20-18	287	5.2907	285	11.169	281	4.1448
dj20-19	305	4.9366	299	10.991	285	7.6422
dj20-20	266	8.5449	260	12.717	259	8.5794
dj50-1	394	107.33	379	258.25	366	138.34
dj50-2	415	167.17	417	295.56	405	67.251
dj50-3	473	90.261	384	152.86	391	63.558
dj50-4	371	73.759	384	195.15	378	206.53
dj50-5	405	101.27	411	170.96	395	94.763
dj50-6	419	95.700	410	221.56	403	54.913
dj50-7	492	122.82	409	249.25	400	111.79
dj50-8	412	116.89	418	359.59	409	103.16
dj50-9	439	92.174	410	198.48	416	146.66
dj50-10	403	163.82	403	347.74	409	140.32
dj50-11	449	223.43	425	397.01	423	89.458
dj50-12	361	174.35	366	439.49	360	81.265
dj50-13	427	165.88	414	476.47	425	71.008
dj50-14	398	80.177	394	353.52	386	178.19
dj50-15	413	69.415	411	200.19	413	78.691

	GRASP C1		GRASP C2		GRASP C3	
Datos	Makespan	Tiempo	Makespan	Tiempo	Makespan	Tiempo
dj50-16	367	133.39	332	485.15	344	121.70
dj50-17	440	104.63	392	385.19	411	93.416
dj50-18	400	143.41	378	481.47	378	122.08
dj50-19	436	86.525	393	204.96	397	125.07
dj50-20	358	62.621	353	225.08	362	127.11

Tabla 9: Resultados GRASP. 20 y 50 trabajos

Como se ha comentado ya anteriormente, los tiempos observados en la Tabla 9 no son comparables, ya que el número de iteraciones no es el mismo, se puede ver cómo hay casos en *GRASP C1* en el que un conjunto de datos con 20 instancias (dj20-1) tarda 11 segundos, y otro con las mismas instancias (dj20-10) tarda 4 segundos, esto es debido a la restricción que se añadió de la estabilidad de la solución, (si hay cien iteraciones sin actualizar, se detiene el algoritmo). El primer conjunto realizó muchas más iteraciones que el segundo.

Estos resultados recogidos en la Tabla 9 son además aleatorios, por lo que, si se volviese a ejecutar, se obtendrían resultados diferentes (aunque similares), y dependiendo de la suerte o mala suerte se llegará a una solución mejor o peor.

Por esa razón, es importante la comparación mediante rankings, ya que así se evita comparar el *makespan* de diferentes conjuntos de datos. Además, podría ser posible que un algoritmo funcione mejor debido a la aleatoriedad en esa ejecución. Sin embargo, si funciona mejor en muchos conjuntos de datos, ya no es cuestión de suerte.

Los rankings obtenidos son:

Algoritmo	**Ranking**
GRASPC1	2.61
GRASPC2	1.95
GRASPC3	1.44

Tabla 10: Rankings GRASP

Observando la Tabla 10 se puede ver cómo el criterio tres es el que mejor funciona, seguido del dos y, finalmente, el uno.

Primero se aplica el Test de Friedman para evaluar si hay diferencias significativas entre los algoritmos. El estadístico obtenido es 28.658, con un p-valor de

5.98e-07. Lo que conduce a rechazar la hipótesis nula, por lo que sí existen diferencias significativas en el *makespan* de los algoritmos.

Con el ANOVA de medidas repetidas se llega a la misma conclusión, ya que se obtiene un p-valor de 4.24e-12, por lo que se puede concluir que sí existen diferencias significativas entre los algoritmos. Se tiene el mismo problema, se violan ciertos supuestos. Esta violación puede afectar a la validez del p-valor obtenido, provocando que se subestime, es decir, que el valor sea artificialmente más pequeño y muestre una mayor significación estadística de la que realmente hay.

Como hay diferencias significativas, se aplica el test de Nemenyi para realizar las comparaciones por pares, obteniendo:

- GRASPC1 vs. GRASPC2: p-valor = 0.0086.
- GRASPC1 vs. GRASPC3: p-valor = 4.4e-07.
- GRASPC2 vs. GRASPC3: p-valor = 0.0569.

Por lo tanto, sí que existen diferencias significativas ($\alpha = 0{,}05$) entre el GRASPC1 y GRASPC2, y entre GRASPC1 y GRASPC3. Sin embargo, con $\alpha = 0{,}05$ no hay diferencias entre GRASPC2 y GRASPC3, pero si se usa un $\alpha = 0{,}06$, sí que se podría decir que hay diferencias significativas entre todos los algoritmos.

Como conclusión final, la metaheurística GRASP que mejor funciona es con el criterio C3, luego el criterio C2 y finalmente la que peor funciona es con el criterio C1. Habiendo en todas ellas diferencias significativas a un nivel $\alpha = 0{,}06$. Pese a que los tiempos no se pueden comparar como ya se explicó anteriormente, analizando los resultados de la tabla se pueden observar unas conclusiones similares a las obtenidas al comparar los tiempos de los criterios Greedy.

Algoritmo	**Media de Tiempo**
GRASPC1_Tiempo	62.5
GRASPC2_Tiempo	158
GRASPC3_Tiempo	58.7

Tabla 11: Media de tiempo GRASP

Viendo la media de lo que tardan los diferentes criterios GRASP en la Tabla 11. GRASP con el criterio dos es la que más tarda con mucha diferencia. Entre GRASP con criterio uno o dos, son similares y no se puede llegar a ninguna conclusión, ya que esos tiempos dependen mucho del límite de cien iteraciones sin modificaciones.

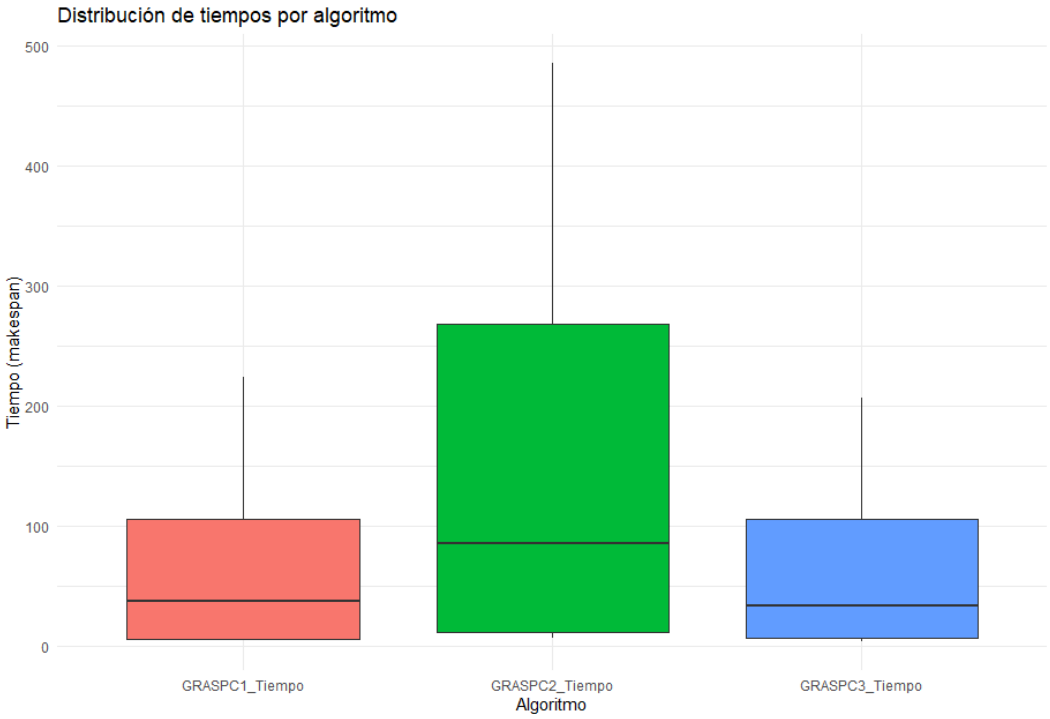

Figura 8: Distribución Tiempos GRASP

La Figura 8 permite ver gráficamente las diferencias de tiempo de cómputo en los diferentes algoritmos GRASP. Se aprecia claramente que el algoritmo GRASP_C2 no solo presenta una media de tiempo bastante superior en comparación con el resto, sino que también exhibe una variabilidad mayor, llegando en algunas instancias a alcanzar tiempos de cómputo demasiado altos en comparación con el resto de criterios utilizados.

Por lo tanto, aunque lo mejor sería probar siempre todos los criterios posibles y ver cuál se adapta mejor a ese caso particular de la solución, se podría decir que el criterio C3 es el mejor, no solo en el *makespan*, sino también en el tiempo de ejecución.

8. Comparación de heurística Greedy y metaheurísticas GRASP de solución al problema DJSSP

Hasta ahora solo se han comparado los criterios greedy entre sí, y los criterios GRASP entre sí. Ahora se va a realizar una comparación entre todos estos algoritmos comentados previamente.

La comparación se realizará mediante el cálculo del RPD, *Relative Percentage Deviation*, aunque esta medida se suele usar para ver cuán diferentes son los números de la media [20], se usará para ver la calidad de las soluciones de los diferentes algoritmos en relación a la mejor solución encontrada.

Se calcula de la siguiente manera:

$$RPD = \frac{Solución - MejorSolución}{MejorSolución}$$

Donde *Solución* es el valor del algoritmo evaluado y *MejorSolución* es el mejor valor obtenido entre todos los algoritmos.

Las ventajas del RPD son:

- Normaliza los resultados para la comparación cruzada.
- Cuantifica el porcentaje de desviación respecto a la solución óptima.
- Es ampliamente usado en literatura metaheurística.

El primer paso es encontrar la mejor solución y luego calcular el RPD para cada algoritmo. Se obtiene la siguiente tabla:

Datos	Mejor solución	Algoritmo mejor solución	RPD GreedyC1	RPD GreedyC2	RPD GreedyC3
dj20-1	276	GRASPC3	0.2644	0.3478	0.2500
dj20-2	282	GRASPC3	0.4255	0.3475	0.1985
dj20-3	267	GRASPC2C3	0.5280	0.1985	0.14602
dj20-4	280	GRASPC3	0.2321	0.2464	0.2142
dj20-5	274	GRASPC2	0.2335	0.2554	0.2773
dj20-6	272	GRASPC3	0.2794	0.3272	0.3051
dj20-7	258	GRASPC3	0.8217	0.1705	0.2170
dj20-8	265	GRASPC2	0.3283	0.2603	0.1811
dj20-9	258	GRASPC1	0.3643	0.2713	0.1627
dj20-10	302	GRASPC3	0.3841	0.3344	0.2384

Datos	Mejor solución	Algoritmo mejor solución	RPD GreedyC1	RPD GreedyC2	RPD GreedyC3
dj20-11	254	GRASPC3	0.2834	0.2519	0.2795
dj20-12	229	GRASPC3	0.5851	0.4148	0.2052
dj20-13	262	GRASPC3	0.4885	0.4541	0.2061
dj20-14	261	GRASPC3	0.3946	0.4061	0.1532
dj20-15	249	GRASPC3	0.3373	0.3012	0.2690
dj20-16	265	GRASPC3	0.2490	0.3056	0.2528
dj20-17	250	GRASPC3	0.4360	0.1400	0.1800
dj20-18	281	GRASPC3	0.2170	0.2811	0.1743
dj20-19	285	GRASPC3	0.2421	0.3017	0.1157
dj20-20	259	GRASPC3	0.4285	0.2355	0.2471
dj50-1	366	GRASPC3	0.1912	0.2185	0.1939
dj50-2	405	GRASPC3	0.3382	0.1703	0.2691
dj50-3	384	GRASPC2	0.4140	0.1718	0.1770
dj50-4	371	GRASPC1	0.3369	0.1805	0.1671
dj50-5	395	GRASPC3	0.3594	0.1974	0.1417
dj50-6	403	GRASPC3	0.2208	0.1637	0.1538
dj50-7	400	GRASPC3	0.7150	0.1925	0.2225
dj50-8	409	GRASPC3	0.2836	0.2909	0.1491
dj50-9	410	GRASPC2	0.2926	0.0878	0.1390
dj50-10	403	GRASPC1C2	0.1885	0.1439	0.1066
dj50-11	423	GRASPC3	0.3333	0.1087	0.1276
dj50-12	360	GRASPC3	0.2111	0.1916	0.1583
dj50-13	414	GRASPC2	0.3574	0.1207	0.1521
dj50-14	386	GRASPC3	0.3704	0.1891	0.1917
dj50-15	411	GRASPC2	0.2871	0.1240	0.1265
dj50-16	332	GRASPC2	0.2590	0.1024	0.1626
dj50-17	392	GRASPC2	0.5408	0.1785	0.2066
dj50-18	378	GRASPC2C3	0.5052	0.2671	0.0952
dj50-19	393	GRASPC2	0.6259	0.1195	0.1170
dj50-20	353	GRASPC2	0.3286	0.1643	0.0793

Tabla 12: RPD Greedy

Datos	Mejor solución	Algoritmo mejor solución	RPD GRASPC1	RPD GRASPC2	RPD GRASPC3
dj20-1	276	GRASPC3	0.0543	0.0398	0.0
dj20-2	282	GRASPC3	0.0496	0.0212	0.0
dj20-3	267	GRASPC2C3	0.0561	0.0	0.0
dj20-4	280	GRASPC3	0.0142	0.0142	0.0
dj20-5	274	GRASPC2	0.0547	0.0	0.0145
dj20-6	272	GRASPC3	0.0330	0.0183	0.0
dj20-7	258	GRASPC3	0.0348	0.0310	0.0
dj20-8	265	GRASPC2	0.0075	0.0	0.0150
dj20-9	258	GRASPC1	0.0	0.0387	0.0232
dj20-10	302	GRASPC3	0.0364	0.0496	0.0
dj20-11	254	GRASPC3	0.0748	0.0078	0.0
dj20-12	229	GRASPC3	0.0698	0.0917	0.0
dj20-13	262	GRASPC3	0.1106	0.0343	0.0
dj20-14	261	GRASPC3	0.0613	0.0306	0.0
dj20-15	249	GRASPC3	0.0682	0.0401	0.0
dj20-16	265	GRASPC3	0.0792	0.0603	0.0
dj20-17	250	GRASPC3	0.0360	0.0520	0.0
dj20-18	281	GRASPC3	0.0213	0.0142	0.0
dj20-19	285	GRASPC3	0.0701	0.0491	0.0
dj20-20	259	GRASPC3	0.0270	0.0038	0.0
dj50-1	366	GRASPC3	0.0765	0.0355	0.0
dj50-2	405	GRASPC3	0.0246	0.0296	0.0
dj50-3	384	GRASPC2	0.2317	0.0	0.0182
dj50-4	371	GRASPC1	0.0	0.0350	0.0188
dj50-5	395	GRASPC3	0.0253	0.0405	0.0
dj50-6	403	GRASPC3	0.0397	0.0173	0.0
dj50-7	400	GRASPC3	0.2300	0.0225	0.0
dj50-8	409	GRASPC3	0.0073	0.0220	0.0
dj50-9	410	GRASPC2	0.0707	0.0	0.0146
dj50-10	403	GRASPC1C2	0.0	0.0	0.0148
dj50-11	423	GRASPC3	0.0614	0.0047	0.0
dj50-12	360	GRASPC3	0.0027	0.0166	0.0
dj50-13	414	GRASPC2	0.0314	0.0	0.0265
dj50-14	386	GRASPC3	0.0310	0.0207	0.0
dj50-15	411	GRASPC2	0.0048	0.0	0.0048

Datos	Mejor solución	Algoritmo mejor solución	RPD GRASPC1	RPD GRASPC2	RPD GRASPC3
dj50-16	332	GRASPC2	0.1054	0.0	0.0361
dj50-17	392	GRASPC2	0.1224	0.0	0.0484
dj50-18	378	GRASPC2C3	0.0582	0.0	0.0
dj50-19	393	GRASPC2	0.1094	0.0	0.0101
dj50-20	353	GRASPC2	0.0141	0.0	0.0254

Tabla 13: RPD GRASP

Una vez obtenidos estos resultados de las tablas 12 y 13, para obtener conclusiones se obtienen los resúmenes de los datos.

Algoritmo	**Media RPD**	**Max RPD**	**Min RPD**
GRASPC1	0.0551	0.2317	0.0
GRASPC2	0.0210	0.0917	0.0
GRASPC3	0.0067	0.0484	0.0
GreedyC1	0.3670	0.8217	0.1885
GreedyC2	0.2309	0.4541	0.0878
GreedyC3	0.1852	0.3051	0.0793

Tabla 14: Resúmenes RPD

En primer lugar, interpretando la Tabla 14, se observa que todos los GRASP funcionan mejor que el resto en al menos un conjunto de datos; sin embargo, los greedy nunca funcionan mejor que los GRASP. Además, todos los GRASP tienen un RPD medio más pequeño que los greedy.

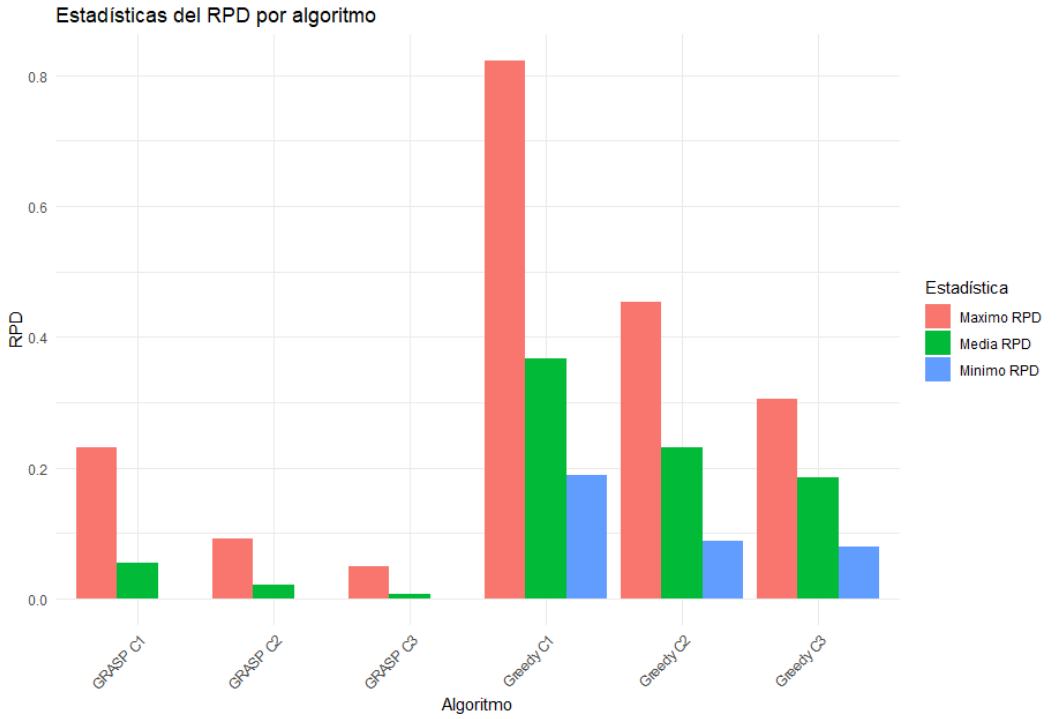

Figura 9: Estadísticas del RPD por Algoritmo

La Figura 9 permite comparar gráficamente los resultados de los estadísticos de la Tabla 14.

Destacar que GRASPC3 es el mejor, con un RPD medio de 0.0067, lo que significa que en promedio sus soluciones están un 0.67 % por encima de la mejor, aunque analizando las tablas completas de arriba, se ve cómo GRASPC3 suele ser la que genera la mejor solución (RPD = 0).

Greedy tiene un rendimiento significativamente peor, siendo GreedyC1 el peor en promedio con un RPD de 0.3670, llegando a ser de 0.8217 en el peor caso, lo que implica que las soluciones son un 36.7 % peores que la mejor solución en cada caso.

Los otros dos greedy tienen también unos valores elevados del RPD, pero GreedyC3 es el mejor de los tres criterios con un RPD de 0.1852.

Hay que analizar también los valores extremos; el peor caso para GRASPC2 sigue siendo bastante bajo, no llega a ser ni un 10 % peor que la mejor solución, por lo que se puede considerar a GRASPC2 y a GRASPC3 como métodos consistentes.

9. Conclusiones

El trabajo realizado ha permitido analizar y comparar diferentes algoritmos para la resolución del problema *Distributed Job-Shop Scheduling Problem*, centrándose en la eficiencia y la calidad de las soluciones obtenidas mediante distintos criterios Greedy y diferentes criterios de la metaheurística GRASP.

Hay que tener en cuenta que las limitaciones del ordenador donde se ha desarrollado este trabajo, por esa razón solo se han podido probar con conjuntos de instancias de tamaño 20 y 50, por lo que aunque pueda parecer que los tiempos de cómputo son similares, si hubiese instancias de tamaño 200 serían inviables algunos de los criterios. Es por esa razón por la que hay que valorar el tiempo de cómputo; si la rapidez fuera crítica, habría que seleccionar criterios que se adaptasen a estas restricciones aunque pudiesen generar peores soluciones.

Por un lado, el criterio Greedy C1 de solución ha sido el menos eficiente en términos del *makespan*, lo que indica que no es una opción recomendable para abordar el problema.

Otra parte importante es la diferencia significativa en el rendimiento de los criterios Greedy, donde el criterio C3 ha demostrado ser la mejor opción en cuanto a la relación *makespan* y tiempo de ejecución. Aunque el criterio C2 también ofrece buenas soluciones, tiene un tiempo de cómputo mucho mayor, lo que podría hacerlo inviable en escenarios donde la rapidez sea crítica.

En cuanto al GRASP, el mejor rendimiento se obtiene con el criterio C3, seguido muy de cerca del criterio C2. Aunque en este caso los tiempos de ejecución no son directamente comparables debido a las diferencias en su implementación, los resultados reflejan tendencias similares a las observadas en los criterios Greedy.

Otro aspecto importante ha sido la comparación entre los algoritmos Greedy y GRASP. Los resultados muestran que GRASP supera en rendimiento a cualquier enfoque Greedy; esto queda evidenciado en los valores del RPD, donde GRASP con el criterio C3 ha demostrado ser el más eficiente, con un RPD medio de 0.0067, consolidándose como el método más confiable y efectivo dentro de todos los analizados en este trabajo.

Por otro lado, los métodos Greedy presentan un rendimiento bastante peor en comparación con los GRASP. Siendo el Greedy C1 el peor, sus soluciones pueden ser de media un 36.7 % peores que la mejor solución posible. Aunque Greedy C3 ha sido el mejor enfoque Greedy, sigue estando considerablemente por debajo de cualquier resultado obtenido con la metaheurística GRASP.

Además del análisis promedio, se han evaluado los casos extremos, donde se comprueba que el algoritmo GRASP usando el criterio C2 o el criterio C3 son métodos consistentes, ya que incluso en los peores casos, las soluciones no han sido un

10 % peores que la mejor solución. Esto sugiere que no solo ofrecen buenos resultados promedios, sino que también mantienen una estabilidad aceptable en escenarios que pudiesen ser adversos.

Como conclusión final, la metaheurística GRASP supera ampliamente al Greedy en términos de la calidad de la solución, ya que se han obtenido soluciones mucho mejores en todas las instancias probadas. Sin embargo, también se ha visto cómo el tiempo de ejecución sigue siendo un factor determinante a la hora de elegir un método de optimización. Aunque GRASP con el criterio C3 ha demostrado ser el mejor en precisión, si hubiese una situación donde el tiempo es una restricción crítica, podría ser necesario considerar otras alternativas rápidas como Greedy con el criterio C3, aunque se sacrificaría la calidad de la solución en beneficio del tiempo de ejecución.

Como posibles líneas de investigación futura se podrían explorar variantes de GRASP con otros algoritmos de optimización como *Simulated Annealing*, *Tabu Search*, o algoritmos genéticos para mejorar aún más la eficiencia. Además, el estudio podría ampliarse considerando nuevas restricciones adicionales.

Referencias

[1] R. Mencía Cascallana et al., «Metaheurística para problemas de Scheduling con múltiples recursos,» 2017.

[2] J. E. Kelley Jr y M. R. Walker, «Critical-path planning and scheduling,» en *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, 1959, págs. 160-173.

[3] F. Patrick Weaver y P. FCIOB, «A Brief History of Scheduling-Back to the Future,» 2014.

[4] D. Pandolfi et al., «Metaheurísticas aplicadas de problemas de scheduling con restricciones,» en *XX Workshop de Investigadores en Ciencias de la Computación (WICC 2018, Universidad Nacional del Nordeste).*, 2018.

[5] H. Xiong, S. Shi, D. Ren y J. Hu, «A survey of job shop scheduling problem: The types and models,» *Computers & Operations Research*, vol. 142, pág. 105 731, 2022.

[6] J. Xie, X. Li, L. Gao y L. Gui, «A hybrid genetic tabu search algorithm for distributed flexible job shop scheduling problems,» *Journal of Manufacturing Systems*, vol. 71, págs. 82-94, 2023.

[7] M. Kurz y R. Askin, «Heuristic scheduling of parallel machines with sequence-dependent set-up times,» *International journal of production research*, vol. 39, n.º 16, págs. 3747-3769, 2001.

[8] J. C. Yepes Borrero, «Una Metaheurística para un problema de secuenciación con necesidad de recursos en los ajustes,» 2017.

[9] B. Roy y B. Sussman, *Les probl emes d'ordonnancement avec contraintes disjonctives, Note DS no. 9 bis, SEMA, Paris*, 1964.

[10] E. Balas, «Machine sequencing via disjunctive graphs: an implicit enumeration algorithm,» *Operations research*, vol. 17, n.º 6, págs. 941-957, 1969.

[11] M. Friedman, «The use of ranks to avoid the assumption of normality implicit in the analysis of variance,» *Journal of the american statistical association*, vol. 32, n.º 200, págs. 675-701, 1937.

[12] W. J. Conover, *Practical nonparametric statistics*. john wiley & sons, 1999.

[13] S. E. Maxwell, H. D. Delaney y K. Kelley, *Designing experiments and analyzing data: A model comparison perspective*. Routledge, 2017.

[14] P. B. Nemenyi, *Distribution-free multiple comparisons.* Princeton University, 1963.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest y C. Stein, «Introduction to Algorithms (3-rd edition),» *MIT Press and McGraw-Hill*, 2009.

[16] A. efficient probabilistic set covering heuristic is presented. The heuristic is evaluated on empirically difficult to solve set covering problems that arise from Steiner triple systems. The optimal solution to only a few of these instances is known. The heuristic provides these solutions as well as the best known solutions to all other instances attempted., *A probabilistic heuristic for a computationally difficult set covering problem - ScienceDirect*, [Online; accessed 2025-03-02]. dirección: https://www.sciencedirect.com/science/article/abs/pii/0167637789900023?via%3Dihub.

[17] *Greedy Randomized Adaptive Search | Algorithm Afternoon*, [Online; accessed 2025-01-20]. dirección: https://algorithmafternoon.com/stochastic/greedy_randomized_adaptive_search/.

[18] *Untitled*, [Online; accessed 2025-01-20]. dirección: https://biblus.us.es/bibing/proyectos/abreproy/3888/fichero/cap3.pdf.

[19] P. Festa y M. G. Resende, «GRASP: basic components and enhancements,» *Telecommunication Systems*, vol. 46, págs. 253-271, 2011.

[20] Admin, *Relative standard deviation formula definition | Relative standard deviation calculator*, [Online; accessed 2025-01-28], dic. de 2021. dirección: https://byjus.com/relative-standard-deviation-formula/.

Anexos

Programación

Para poder realizar todos los análisis comparativos, se desarrollan implementaciones de los algoritmos analizados anteriormente. La programación se realizó con *FICO Xpress Mosel*, un lenguaje de modelado especializado en la formulación y resolución de problemas de optimización.

Para la implementación de los algoritmos, se siguen los pseudocódigos explicados anteriormente, teniendo cuidado con las variables inicializadas.

La parte más importante de la programación es una función que permite añadir un trabajo a una fábrica. Aunque en un principio esto pudiera sonar sencillo, se complica debido a la gran cantidad de restricciones que hay que tener en cuenta para saber en qué momento de tiempo se inició un trabajo y cuándo finaliza.

Este procedimiento tiene como objetivo introducir un trabajo dado en una fábrica dada, respetando los huecos libres y las restricciones. Para ello, se siguen los siguientes pasos.

1. Inicialización. Se identifica la primera máquina donde debe empezar el trabajo a procesar, también se buscar el tiempo más temprano en el que el trabajo puede empezar a ser programado.

2. Recorrer las máquinas. Se recorren las máquinas por las que debe pasar el trabajo en el orden de procesamiento.

3. Buscar los huecos libres. Para cada máquina se realiza el siguiente procedimiento:

   - Se busca un trabajo ya asignado que tenga el inicio más temprano.
   - Si no hay trabajos previos o ya se han recorrido todos se programa el trabajo en el primer hueco disponible que se buscó antes.
   - Si hay trabajos, se compara el espacio entre los dos trabajos (Tiempo). Si el espacio es suficiente, el trabajo puede realizarse su procesamiento en la máquina en ese momento. En caso contrario, se actualiza el tiempo más temprano en el que el trabajo puede empezar a ser programado para poder seguir buscando.

4. Actualización de tiempos. Se actualizan los tiempos de inicio y de fin del trabajo en la máquina. Se avanza a la siguiente máquina teniendo en cuanta

que el tiempo más temprano es el de finalización de máquina anterior. Cuando no haya más máquinas se actualiza el tiempo final de la solución, tomando el tiempo mayor de finalización.

Debido a la complejidad de este procedimiento, es muy importante poder comprobar que las soluciones son correctas; para ello, se exportan los resultados a un archivo de texto. Este archivo contiene las variables principales del problema, la lista ordenada de trabajos procesados y la lista de trabajos, máquinas y fábricas con su tiempo de inicio y finalización.

Ese fichero se lee con *Python* y utilizando librerías como *pandas* y *Matplotlib* se crea una interfaz gráfica que permite visualizar progresivamente los trabajos programados. De esta forma se van mostrando los trabajos progresivamente, facilitando así la comprobación de que tanto los algoritmos como la función explicada anteriormente funcionan a la perfección.