Proyecto Fin de Grado
Grado en Ingeniería de Organización Industrial

Algoritmo de búsqueda local iterada para la secuenciación en talleres de flujo y minimización de makespan.

Autora: Ángela León Martínez
Tutor: Víctor Fernandez-Viagas Escudero

Dpto. Organización Industrial y Gestión de Empresas I
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018

Proyecto Fin de Grado
Grado en Ingeniería de Organización Industrial

Algoritmo de búsqueda local iterada para la secuenciación en talleres de flujo y minimización de makespan.

Autora:
Ángela León Martínez

Tutor:
Víctor Fernandez-Viagas Escudero

Dpto. de Organización Industrial y Gestión de Empresas I
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2018

Trabajo de Fin de Grado: Algoritmo de búsqueda local iterada para la secuenciación en talleres de flujo y minimización de makespan.

Autora: Ángela León Martínez
Tutor: Víctor Fernández-Viagas Escudero

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2018

El Secretario del Tribunal

A los que estuvieron
A los que están
A los que estarán

Agradecimientos

En primer lugar, me gustaría agradecer a mis padres por apoyarme y permitirme la oportunidad de formarme.

También me gustaría agradecer a todos mis amigos el apoyo recibido durante la realización de este proyecto.

Por último, me gustaría agradecer a mi tutor su apoyo y comprensión a lo largo de todo el proyecto.

Ángela León Martínez

Resumen

En este documento se abordará la secuenciación de la producción en un entorno de *assembly* (ensamblado) para una empresa real dedicada a la fabricación de moldes para envases plásticos. Este entorno, tiene la particularidad de contar con la restricción *missing operations*, lo que quiere decir, que no todos los trabajos pasan por todas las máquinas. Debido a la complejidad del problema (*NP-hard*), éste será resuelto mediante métodos de resolución aproximados. En concreto, se ha recurrido a metaheurísticas basadas en trayectoria para su resolución. De entre todas esta metaheurísticas se ha seleccionado la conocida como *Iterated Local Search*, ILS, o Búsqueda Local Iterada. Se han desarrollado dos variables de la metaheurística para la resolución del problema. Ambas han sido implementadas mediante programación en C.

El objetivo de este proyecto es, por tanto, la creación del algoritmo que nos permita definir una buena secuencia de entrada de los trabajos en un tiempo de computación razonable, de manera que se minimice el *makespan*, o tiempo de finalización máximo, de la producción de la empresa. Las dos variantes del algoritmo ILS, se compararán con el fin de identificar su eficiencia en los diferentes escenarios. Para ello, se creará una batería de problemas y se analizarán los resultados obtenidos en todas ellas.

La batería de problemas ha sido creada aleatoriamente, combinando distintos valores de parámetros como son el número de etapas, trabajos y porcentaje de missing. Para cada combinación se han generado diez instancias aleatorias formando un total de 360 problemas estudiados.

El siguiente documento se encuentra estructurado de la siguiente manera:

En el capítulo 1, se realiza una introducción general a la programación y secuenciación de la producción. También se explican los conceptos básicos necesarios para entender el problema descrito y los métodos empleados para su resolución. En este capítulo se explican los diferentes entornos de producción existentes, las distintas FO que pueden existir para cada problema y sus distintas restricciones.

En el capítulo 2, se realiza una descripción del problema. En primer lugar, se describe el entorno productivo de la empresa real, indicando datos como tiempos de proceso y rutas de cada trabajo. Posteriormente, se realiza una descripción formal del problema y un modelado de éste. Lo que nos permite asemejarlo con alguno de los entornos comentados en el capítulo 1.

Debido a la complejidad del problema (*NP-hard*) se recurren a metaheurísticas aproximadas para su resolución. En el capítulo 3, se realiza una descripción general sobre las metaheurísticas más

populares, clasificándolas en metaheurísticas basadas en trayectoria y metaheurísticas basadas en población. En el capítulo 4, puede encontrarse una descripción más detallada sobre la metaheurística *Iterated Local Search (ILS)* y sobre las dos variantes que han sido empleadas para la resolución del problema.

En el capítulo 5, puede encontrarse cómo se ha realizado la implementación de esta metaheurística para la resolución del problema mediante programación en C. En el capítulo 6 pueden encontrarse los resultados computacionales de la implantación de esta metaheurística, además de una comparación sobre la eficacia y eficiencia de los algoritmos en determinados escenarios.

Índice

Agradecimientos **vii**
Resumen **ix**
Índice **xi**
Índice de Tablas **xiii**
Índice de Figuras **xvi**
1 Introducción **1**
1.1 Notación *3*
1.1.1. Clasificación de los modelos de programación de la producción 5
2 Descripción del problema **9**
2.1. Descripción de la empresa *9*
2.2. Descripción formal del problema *11*
2.2.1. Datos del problema 11
2.2.2. Modelado del problema 13
3. Metaheurísticas propuestas **16**
3.1. Clasificación de las metaheurísticas *16*
3.1.1. Metaheurísticas basadas en trayectoria 17
3.1.2. Metaheurísticas basadas en población 18
4. Iterated Local Search (ILS) **20**
4.1. Algoritmo Iterated Local Search 1 *22*
4.2. Algoritmo Iterated Local Search 2 *23*
5. Implementación **24**
5.1. Versión 1 del Algoritmo ILS *24*
5.2. Versión 2 del Algoritmo ILS *26*
5.3. Resolución de las instancias *28*
5.3.1. Generación de batería de problemas 29
5.3.2. Resolución del problema inicial 32
6. Resultados computacionales **34**
6.1. Evaluación computacional en batería de problemas *35*
6.1.1. Generación de problemas 35
6.1.2. Análisis de resultados en función del tiempo 37
6.1.3. Análisis de resultados en función del número de iteraciones 39
6.1.4. Análisis de resultados en función de la FO 42
6.2. Análisis de resultados en el problema estudiado *46*
7. Conclusiones **48**
8. Bibliografía **50**
8. Anexos **52**
8.1. Código en C *52*

ÍNDICE DE TABLAS

Figura 1. APS. Sistema de Planificación Avanzada. 2
Figura 2. APS. Sistema de Planificación Avanzada. 3
Figura 3. Cadena de precedencia. 6
Figura 4. Precedencias outtree. 7
Figura 5. Precedencias intree. 7
Figura 6. Precedencias árbol. 7
Figura 7. Layout del entorno de ensamblado del problema. 14
Figura 8. Clasificación de metaheurísticas. 17
Figura 9. Intercambio adyacente. 21
Figura 10. Intercambio. 21
Figura 11. Inserción. 21
Figura 12. Tiempo en ms en función del número de máquinas 37
Figura 13. Tiempo en ms en función del número de trabajos 38
Figura 15. Tiempo en ms en función del % missing 38
Figura 16. Número de iteraciones en función del número de máquinas 39
Figura 17. Número de iteraciones en función del número de trabajos 40
Figura 18. Número de iteraciones en función del número del % de missing 40
Figura 19. ARPD en función del número de trabajos 42
Figura 20. ARPD en función del número de máquinas 43
Figura 21. ARPD en función del número del porcentaje de missing. 44

ÍNDICE DE FIGURAS

Tabla 1. Rj del problema 12
Tabla 2. Tiempos de proceso pij expresados en horas. 12
Tabla 3. Tiempos de proceso sij expresados en horas. 13
Tabla 4. Problemas generados 36
Tabla 5. Número medio de iteraciones hasta alcanzar una solución. 39
Tabla 6. ARPD en función del número de trabajos. 42
Tabla 7. ARPD en función del número de máquinas. 43
Tabla 8. ARPD en función del número del porcentaje de missing. 44
Tabla 9. ARPD promedio de ambos algoritmos 45

1 INTRODUCCIÓN

"Reunirse es un comienzo, permanecer juntos es el progreso y trabajar juntos es el éxito".
Henry Ford

LA Programación de la Producción puede definirse como el conjunto de técnicas o conocimientos cuyo objetivo es la asignación de recursos a tareas. Cada una de estas tareas debe ser procesada en el lugar y en el momento correcto de manera que se obtenga la máxima eficacia respecto a uno o más objetivos que se deseen optimizar, así como la máxima eficiencia de los recursos. El Control de la Producción es el conjunto de herramientas utilizadas para realizar el seguimiento del programa de producción y ejecutar acciones correctoras en caso de ser necesario.

La Programación y Control de la Producción, ámbito en el cual se engloba este proyecto, se inscribe a su vez dentro de la Organización de la Producción (*Production Management*).

Tanto los recursos como las tareas pueden variar enormemente en función de la organización donde nos encontremos. Así, podemos identificar como recursos las máquinas en un taller, las pistas en un aeropuerto, las unidades de procesamiento en un entorno informático… Las tareas asociadas con estos recursos pueden ser identificadas respectivamente con operaciones en un proceso productivo, despegues y aterrizajes en un aeropuerto y ejecuciones de programas en equipos informáticos.

No todos los problemas dentro del campo de programación de la producción responderán a los mismos parámetros. Cada tarea puede contar con un nivel de prioridad distinto, con fechas de inicio y finalización que deben cumplirse obligatoriamente o con tiempos de preparación. A su vez, en cada caso, la función objetivo que se trata de optimizar será completamente distinta, pudiendo ser ejemplos de ésta la minimización del tiempo de proceso o del número de trabajos finalizados fuera de plazo.

Para lograr esta correcta asignación de tareas a recursos es necesaria la toma de un gran número de decisiones a lo largo del tiempo, en ámbitos muy distintos y con impactos muy diferentes. Es por ello que se necesita la adopción de una estructura jerárquica para tomar las decisiones.

El sistema de planificación avanzada, APS o *Advanced Planning Systems*, hace referencia al proceso de decisión para la fabricación en el cuál tanto las materias primas como la capacidad de producción son asignadas de manera óptima para satisfacer la demanda. Este sistema, divide las tareas en cuatro grandes áreas: compras, fabricación, distribución y ventas. Dentro de cada área, las decisiones y tareas se dividen en diferentes horizontes temporales según si su impacto será visible a largo plazo, medio plazo o corto plazo. Las decisiones se encuentran relacionadas entre sí al igual que se encuentran relacionadas las cuatro grandes áreas. Toda decisión que sea tomada dentro de estas áreas influye en mayor o menor manera sobre otras. Por ejemplo, la planificación del personal influirá directamente sobre la secuenciación de la producción, ya que esta deberá adaptarse a los recursos disponibles. Todas estas relaciones entre módulos y la distribución temporal y en áreas de las distintas tareas pueden observarse en la Figura 1.

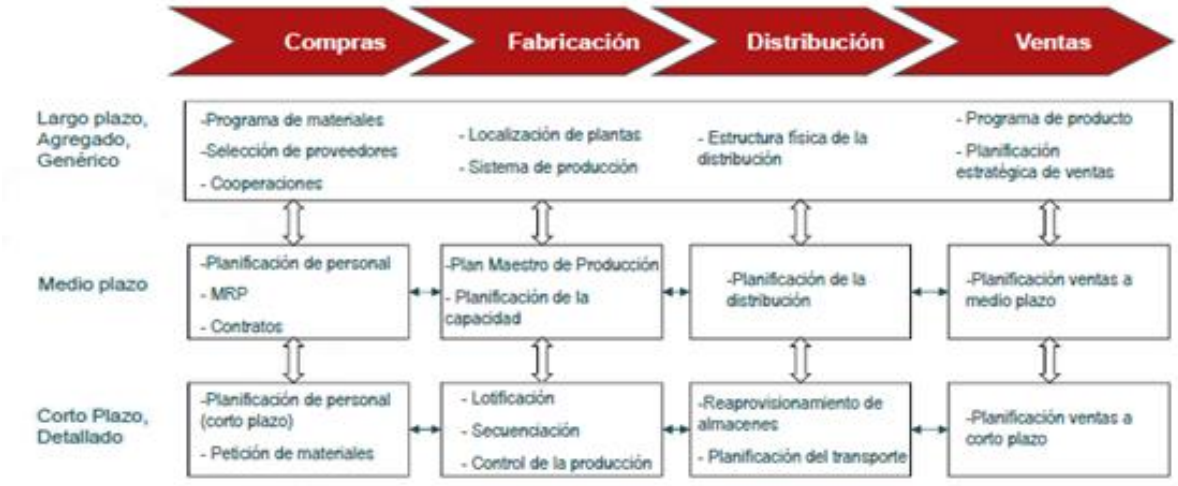

Figura 1: *APS. Sistema de Planificación Avanzada. Fuente: Diapositivas Programación y Control de la Producción [2017].*

El objeto de estudio de este trabajo, tal y como se ha expresado con anterioridad, se inscribe dentro de la Programación y Control de la Producción (*Scheduling & Control*). Como puede observarse en la Figura 2, ésta se engloba dentro del área de fabricación y proviene del Plan Maestro de Producción y de la Planificación de la Producción. A corto plazo, se puede observar que intercambia información con el área de compras para la planificación de las necesidades de

material. También existe un intercambio de información con el área de distribución para la planificación del transporte.

Figura 2: *APS. Sistema de Planificación Avanzada. Fuente: Diapositivas Programación y Control de la Producción [2017].*

1.1 Notación

Para la definición de la notación que será necesario usar de ahora en adelante, se ha tomado como referencia la definida por Pinedo (2010). Se supondrá un número finito tanto de máquinas como de trabajos. El número de trabajos será denotado por n mientras que el número de trabajos se denotará por m. El subíndice j se refiere a trabajos mientras que el subíndice i hace referencia a máquinas. Cuando sea necesario referirse a una operación en una máquina, se usará el conjunto (i, j). A continuación, se indican diferentes datos que pueden ir asociados con el trabajo j:

- **Tiempo de proceso** (p_{ij}): representa el tiempo de proceso del trabajo j en la máquina i. El subíndice i se omite cuando el tiempo de proceso es independiente de la máquina o el trabajo solo puede ser procesado en una máquina.

- **Fecha de llegada** (r_{ij}): representa el instante a partir del cual el trabajo j se encuentra listo para ser procesado en la máquina i.

- **Fecha de entrega** (d_j): representa el instante en el cual el trabajo j debe estar terminado. La finalización después de la fecha de entrega puede violarse en algunas ocasiones, aunque

puede estar sujeta a una penalización. En el caso de que la fecha de entrega sea obligatoria, se denota como $\bar{d}_j$.

- **Peso** (ω_j): representa un indicador para establecer una jerarquía entre el conjunto de trabajos j del modelo.

- **Tiempo de terminación** (C_{ij}): representa el instante en el que el trabajo j finaliza su procesado en la máquina i. Cuando se haga referencia al momento en el cuál el trabajo j finalice en el entorno, es decir, cuando el trabajo haya sido procesado por la última máquina, se denotará como C_j.

- **Tiemplo de flujo** (F_j): representa el tiempo que el trabajo j se encuentra en el entorno. Puede definirse como:

$$F_j = C_j - r_j$$

- **Retraso del trabajo** (L_j): representa la diferencia entre la fecha de entrega y el tiempo de terminación. Será positivo si el trabajo se ha completado antes de su fecha de entrega y negativo en caso contrario. Puede definirse como:

$$L_j = C_j - d_j$$

- **Tardanza del trabajo** (T_j): representa cuanto se ha retrasado un trabajo. En caso de que el trabajo se haya completado antes de su fecha de entrega, su valor será 0. Puede definirse como:

$$T_j = max(0, C_j - d_j) = max(0, L_j)$$

- **Adelanto del trabajo** (E_j): representa cuanto se ha adelantado un trabajo. En caso de que el trabajo se haya completado después de su fecha de entrega, su valor será 0. Puede definirse como:

$$E_j = max(0, -L_j) = max(0, d_j - C_j)$$

- **Trabajo tardío** (U_j): representa si un trabajo ha finalizado después de su fecha de entrega. Puede definirse como:

$$U_j = \begin{cases} 1 \; si \; C_j > d_j \\ 0 \; en \; caso \; contrario \end{cases}$$

1.1.1. Clasificación de los modelos de programación de la producción

Para describir los distintos modelos de programación de la producción, usaremos la notación propuesta por Graham et al. (1979) que, además, es la más extendida en la actualidad. Esta notación, consiste en ir describiendo los modelos en tres campos, siendo cada uno de ellos:

- α: tipo de entorno de fabricación (*layout*). Define la disposición de las máquinas y el número de ellas.
- β: restricciones que caracterizan el sistema productivo.
- γ: objetivo u objetivos a minimizar o maximizar para obtener la solución más eficaz al modelo de producción.

1.1.1.1 Tipos de entornos. Características de las máquinas (α)

La disposición de los recursos es una parte fundamental para la definición y resolución de los problemas de programación de la producción. Podemos identificar siete tipos de entornos:

- ***Single machine*** (1): una única máquina se encarga de procesar todos los trabajos. Es el caso más simple y es un caso especial de otros entornos más complejos.

- ***Identical parallel machines*** (Pm): m máquinas idénticas en paralelo. El trabajo *j* requiere de una única operación y puede ser procesado en cualquiera de las m máquinas. Al ser todas las máquinas idénticas, el tiempo de proceso p_{ij} será el mismo en todas ellas.

- ***Uniform parallel machines*** (Qm): m máquinas en paralelo con distintas velocidades de procesado. La velocidad de cada máquina se denota por v_i. El tiempo de proceso puede calcularse como:

$$p_{ij} = p_j/v_i.$$

- ***Unrelated parallel machines*** (Rm): m máquinas distintas en paralelo, a diferencia del anterior, las velocidades de procesado dependen de cada trabajo. El tiempo de proceso puede calcularse como:

$$p_{ij} = p_j/v_i.$$

- ***Flow shop*** (Fm): m máquinas en serie. Cada trabajo debe ser procesado en cada una de las m maquinas. Todos los trabajos siguen la misma ruta, es decir, deben pasar por las máquinas en el mismo orden.

- ***Job shop*** (Jm): m máquinas. Cada trabajo tiene una ruta predeterminada, sin embargo, esta no tiene que ser la misma para todos ellos. Otra de las diferencias con el entorno *flow shop* es que cada trabajo puede pasar más de una vez por la misma máquina.

- ***Open shop*** (O): m maquinas**.** Cada trabajo debe ser procesado en cada una de las máquinas, sin embargo, puede que alguno de los tiempos de proceso sea cero. No hay restricciones correspondientes a la ruta de cada trabajo.

1.1.1.2. Características de los trabajos. Restricciones (β)

Las características de los trabajos, más conocidas como restricciones, ayudan a definir el entorno según sus características. Las restricciones más comunes son las siguientes:

- **Fechas de llegada** (r_j): esta restricción indica que el trabajo j no puede comenzar a ser procesado antes de su fecha de llegada r_j.

- **Interrupciones** ($prmp$): esta restricción hace referencia a si los trabajos son o no interrumpibles. Existen tres tipos de trabajos basándonos en esta característica:
  - $Pmtn - non - resumable$: no se puede interrumpir una tarea sin perder todo el trabajo.
  - $Pmtn - semi - resumable$: si se interrumpe el trabajo, al retomarlo se reinicia parcialmente, es decir, solo se pierde parte del trabajo.
  - $Pmtn - resumable$: se puede interrumpir el trabajo y se retoma en el momento de la interrupción.

- **Precedencia de los trabajos** ($prec$): esta restricción puede aparecer en los entornos de single machine o parallel machine. Se requiere que uno o más trabajos sean completados antes de que otro pueda comenzar. Existen varios tipos de precedencia:

  - Cadena de precedencia (*chain precedence*): cada trabajo consta de un predecesor y de un sucesor.

Figura 3: Cadena de precedencia. Fuente: Diapositivas Programación y Control de la Producción [2017].

- Precedencias *outtree*: cada trabajo consta de un único predecesor y al menos de un sucesor.

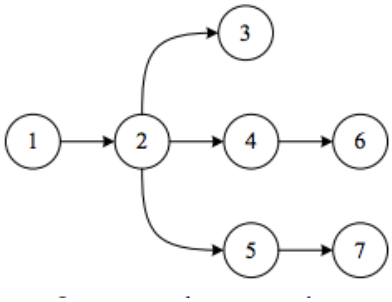

Figura 4: *Precedencias outtree. Fuente: Diapositivas Programación y Control de la Producción [2017].*

- Precedencias *intree*: cada trabajo consta de al menos un predecesor y de un único sucesor.

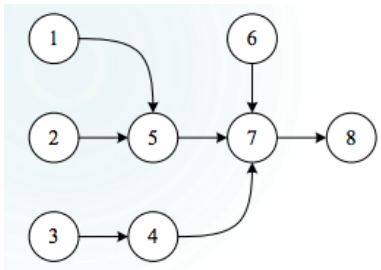

Figura 5: *Precedencias intree. Fuente: Diapositivas Programación y Control de la Producción [2017].*

- Precedencias árbol: no siguen ninguna pauta.

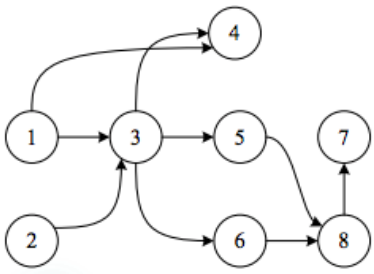

Figura 6: *Precedencias árbol. Fuente: Diapositivas Programación y Control de la Producción [2017].*

- **Tiempos de cambio** (s_{jk}): esta restricción aparece cuando los trabajos necesitan un tiempo adicional de preparación en máquina antes de ser procesados. Los tiempos de setup pueden ser dependientes de la secuencia, en este caso se denotan por s_{ijk}.

- **Secuencia de permutación** ($prmu$): esta restricción puede aparecer únicamente en los entornos flow shop para indicar que cada una de las máquinas opera según la regla de despacho FIFO (*First In First Out*). Esto significa, que el orden en el que los trabajos pasan por la primera máquina se mantiene durante el resto del sistema.

- **Espera no permitida** (nwt): esta restricción indica que una vez que un trabajo ha iniciado a procesarse, debe hacerlo en todas las máquinas sin tiempos de espera entre ellas.

1.1.1.3 Función objetivo (γ)

Todo problema de programación de la producción tiene por objetivo la obtención de la máxima eficacia, minimizando o maximizando uno o varios objetivos. Los más comunes que se tratan de minimizar son los siguientes:

- **Makespan** (C_{max}): es el tiempo máximo de finalización de los trabajos en el entorno. Un makespan bajo implica una buena utilización de la maquinaria.

- **Máximo retraso** (L_{max}): es el tiempo máximo de todos los retrasos.

- **Tiempo de finalización total ponderado** ($\Sigma\omega_j C_j$): es la suma ponderada de los tiempos de finalización. Supone un buen indicador para los gastos de inventario que se están produciendo a causa de la planificación de la producción.

- **Tiempo de tardanza total ponderado** ($\Sigma\omega_j T_j$): es la suma ponderada de los tiempos de tardanza

2 DESCRIPCIÓN DEL PROBLEMA

"Lo más incomprensible acerca de este mundo
es que es comprensible".
Albert Einstein

EN este capítulo se procederá a realizar una descripción de la empresa y de su organización productiva. También se realizará una descripción formal del problema de producción existente en esta. Los datos obtenidos del estudio de la producción en la empresa permitirán el modelado del problema, permitiendo su definición según los parámetros α, β y γ explicados en la sección 1.1.1.

2.1. Descripción de la empresa

Moldes Martínez Puerto S.I. es una empresa familiar fundada en 1977 en la ciudad de Andújar, (Jaén). Desde entonces, la empresa se ha especializado en la fabricación de moldes por soplado, sirviéndose de la última tecnología en fabricación para lograrlo.

A pesar de que durante sus primeros años la empresa tan solo se dedicaba a la fabricación del molde, siendo necesaria la subcontratación del resto de etapas del proceso. En la actualidad, gracias al crecimiento obtenido, se ha logrado la integración del personal y el equipo necesario para la realización del proceso integro de fabricación; desarrollo inicial, diseño, prototipado, fabricación del molde, control y supervisión del producto; son las etapas necesarias hasta la obtención del producto final.

Durante los últimos años, la empresa ha incorporado a su actividad el montaje y reparación de maquinaria en general. Para la realización de esta actividad, se comparten recursos humanos con la línea de producción.

Centrándonos en la línea de fabricación de moldes, proceso en el que se basa este trabajo, podría decirse que la producción se realiza exclusivamente bajo demanda. Las distintas necesidades en cuanto a características y diseño de cada cliente hacen necesaria una fase previa de diseño y prototipado única para cada producto realizado. Por esto, no se puede contemplar la producción por stocks.

Una vez realizadas las etapas iniciales del proceso, se pasa a la fabricación del molde en línea de producción. A pesar de trabajar con distintos diseños, los tiempos de proceso y el proceso de fabricación se mantienen constantes**.**

Por lo tanto, el flujo productivo no comienza hasta que se dan por finalizadas las etapas de diseño y prototipado, el cliente efectúa una orden de compra y se reciben los materiales necesarios para la fabricación.

Los materiales necesarios se obtienen de cuatro proveedores distintos, cada uno de ellos se encarga del envío de los materiales necesarios para formar cada una de las distintas partes del molde, que, tras ser fabricadas, deben pasar por un proceso de ensamblado. Las piezas que conforman el producto son las siguientes:

- Cavidad del molde fabricada en una aleación de alumec 79
- Fondo del molde fabricado en aluminio redondo
- Boca del molde fabricada en acero
- Anclajes fabricados en acero

La maquinaria utilizada en la línea es exclusiva para la fabricación del molde. Sin embargo, tal y como se ha comentado anteriormente, parte del personal que trabaja en la cadena de producción compagina su trabajo con el de montaje y reparación de maquinaria. En caso de que el operario tenga tareas a realizar en ambas áreas, siempre se priorizará la producción en serie debido a su criticidad. Por tanto, aunque los operarios realicen tareas en diferentes proyectos de manera “simultánea”, la línea de producción no sufre ningún retraso por causas debidas a personal humano.

En la línea de producción podemos encontrar ocho máquinas necesarias para la fabricación del producto:

- Sierra de cinta (M1)
- Torno convencional (M2)
- Taladro vertical de columna (M3)
- Fresadora CNC (M4)
- Rectificadora (M5)
- Torno CNC (M6)

- Centro de mecanizado (M7)
- Rotalín eléctrico de pulido (M8)

El transporte de las piezas entre máquinas lo realizan los operadores de las máquinas, las distancias entre las distintas máquinas son despreciables, por lo que no consideraremos tiempos de desplazamientos. El almacenamiento se realiza a pie de máquina, suponiéndose infinito.

La problemática a la que se enfrenta la empresa, proviene de una programación de la producción prácticamente inexistente, donde la producción se viene realizando de manera ineficiente. De manera, que no se alcanzan los objetivos fijados en cuánto a reducción del *makespan*.

La existencia de diferentes tiempos de *set-up* en cada una de las operaciones y el hecho de que no todos los trabajos tengan que ser procesados en todas las máquinas hace que hallar una secuenciación eficiente de las operaciones de manera manual para satisfacer los objetivos de la empresa sea prácticamente imposible.

2.2. Descripción formal del problema

2.2.1. Datos del problema

En este apartado se va a proceder a la explicación de todos los datos recabados sobre la producción en la empresa mediante observación y encuestas a trabajadores. Para la producción del molde, son necesarias la fabricación de cuatro piezas, que más tarde deberán ensamblarse en una única máquina.

El problema de producción en la empresa, tiene como entradas el número de trabajos, número de máquinas, rutas de trabajos, tiempos de proceso y tiempos de set-up.

El conjunto de máquinas se puede definir como $M = \{1, \ldots, m\}$, siendo en este caso particular $m = 8$. De ahora en adelante se usará el índice $i \in M$ para su denotación. El conjunto de trabajos puede definirse como $N = \{1, \ldots, n\}$, siendo en este caso $n = 4$. Los índices usados para la denotación de trabajo serán $j, k \in N$.

La ruta de los trabajos $R = (1, 2, \ldots, n)\ \forall j \in N$, es decir, el orden en el que cada trabajo debe pasar por cada máquina se encuentra predeterminado. Para este problema en concreto, teniendo en cuenta que no todos los trabajos pasarán por todas las máquinas, la tabla de rutas de trabajos es la siguiente:

jobs (*j*)	*Rj*			
1	1	2	6	
2	1	4	5	7
3	1	2	7	
4	3	5	7	
Ensamblado (1,2,3,4)	8			

***Tabla 1: Rj** del problema*

En la tabla 1 se muestran las rutas de los distintos trabajos del problema. En la parte superior de la tabla, se muestra la ruta que debe seguir cada una de las piezas para su correcta fabricación en la primera fase correspondiente al flowshop. En la parte inferior, se muestra la información relativa a la ruta del conjunto de piezas en la fase de *assembly*. Todas las tablas de datos del problema presentan la misma estructura.

Cada uno de los trabajos, consta de una determinada duración en cada máquina. Se denomina p_{ij} al tiempo de procesamiento del trabajo j en la máquina i. El problema que se va a estudiar, consta de los tiempos de procesos p_{ij} expresados en horas y recogidos en la tabla 2.

	jobs (j)				
machine (i)	1	2	3	4	Ensamblado (1,2,3,4)
1	-	2	6	3	-
2	-	4	-	6	-
3	3	-	-	-	-
4	-	-	8	-	-
5	6	-	2	-	-
6	-	-	-	6	-
7	5	6	9	-	-
8	-	-	-	-	8

***Tabla 2:** tiempos de proceso $\boldsymbol{p_{ij}}$ expresados en horas.*

Además, en la fabricación del producto existen tiempos adicionales de preparación en cada máquina a la llegada del trabajo, que se definen como tiempos de set-up. Se denotan por s_{ij} al tiempo de preparación en la máquina i antes de realizar el trabajo j. Los datos de tiempos de set-up expresados en horas para este problema se encuentran en la tabla 3.

	jobs (*j*)				
machine (*i*)	1	2	3	4	Ensamblado (1,2,3,4)
1	-	-	-	-	-
2	-	-	-	-	-
3	1	-	-	-	-
4	-	-	1	-	-
5	-	-	-	-	-
6	-	-	-	-	-
7	3	3	4	-	-
8	-	-	-	-	-

Tabla 3: *tiempos de proceso* s_{ij} *expresados en horas.*

2.2.2. Modelado del problema

Tras analizar el problema y sus datos, observamos que se corresponde con un entorno de secuenciación de ensamblado, dividido en dos fases. La primera de ellas se corresponde con un flowshop sin permutación. La segunda fase puede identificarse como un single machine.

Los entornos de secuenciación de ensamblado se dan en industrias y servicios en los cuales una actividad sólo puede comenzar cuando una serie de tareas previas han sido completadas. Estas tareas deben ser secuenciadas en conjunto, ya que de otra forma no se aseguraría una correcta sincronización y eficiencia. En la actualidad, la mayoría de las compañías se centran en la fabricación de partes específicas de sus productos, externalizando la producción o incluso comprando aquellos componentes que no constituyen sus competencias fundamentales. Todo esto, conlleva a un aumento de la relevancia de este tipo de problemas.

Hay diferentes tipos de *layout* según los entornos que forman cada una de las fases. Se puede observar una representación gráfica del *layout* de ensamblado del problema expuesto en este trabajo en la Figura 7.

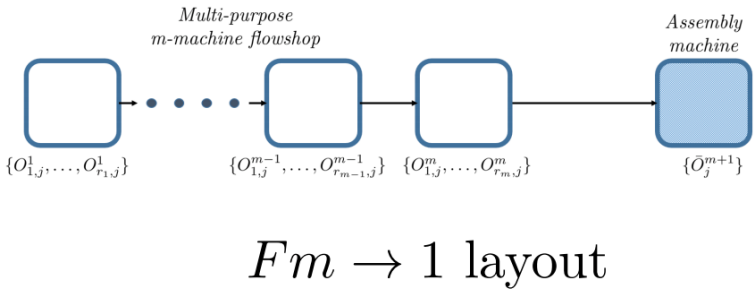

***Figura 7**: layout del entorno de ensamblado del problema. Fuente: Framiñan, Perez-Gonzalez y Fernandez-Viagas [2017]*

No existe una notación clara para este tipo de entornos. Framiñan, Perez-Gonzalez y Fernandez-Viagas (2017), proponen una notación común para los problemas de secuenciación de ensamblado. Esta notación ha sido usada para denotar este problema de la siguiente forma:

$$Fm \rightarrow 1|s_{ij}, prmu, skip\ |C_{\max}$$

Analizando de manera un poco más detallada cada una de las dos fases, se puede observar que la primera de ellas, a pesar del hecho de que no todos los trabajos sean procesados en todas las máquinas, se corresponde con un entorno de flujo regular o flowshop. En este caso, existirían 7 máquinas en serie. Todos los trabajos siguen la misma ruta, es decir, deben procesarse en las máquinas en el mismo orden. Cuando un trabajo ha sido completado en una máquina, se une a la cola de la siguiente máquina. En este caso, el problema planteado se rige por la disciplina (FIFO), es decir, los trabajos serán procesados en máquina por estricto orden de llegada. Esto se refleja en la restricción β de permutación ($prmu$).

En cuanto a las restricciones aplicables a esta primera fase se consideran tiempos de *set-up*, recogidos en la tabla 3 del apartado 2.1, independientes de la secuencia y denotados por s_{ij}. Tambien encontramos la restricción de *missing operations,* la cuál representa que no todos los trabajos pasan por todas las máquinas.

Independientemente de esta restricción, existen una serie de restricciones, que se dan en la mayoría de entornos a no ser que se indique lo contrario. Podrían clasificarse como suposiciones generales y también deben tenerse en cuenta:

- No existe fecha de llegada para los trabajos, todos los trabajos están disponibles al comienzo del horizonte temporal
- Cada máquina sólo puede procesar un trabajo en un instante de tiempo.

- Cada trabajo sólo puede ser procesado en una máquina en cada instante de tiempo.
- Cada trabajo está compuesto por una serie de operaciones que deben ser procesadas en el orden predefinido por la ruta Rj.
- No se permite la interrupción de los trabajos.
- El buffer entre las máquinas se supone infinito.
- Los tiempos de transporte entre máquinas se suponen despreciables.

La segunda fase de la producción correspondería a un entorno single machine. Tras ser completados todos los trabajos en la primera fase, estos pasan a una segunda fase, en la cual se produce el ensamblado de las diferentes piezas en una única máquina.

En resumen, el flujo de productos tendrá una única dirección, todos los productos siguen la misma ruta, es decir, pasan por las máquinas en el mismo orden, aunque algunos trabajos pueden no tener que procesarse en alguna máquina. Una vez hayan sido procesados todos los trabajos de la primera fase, pasan a la segunda correspondiente al Assembly, donde son procesados en conjunto.

En cuanto a la FO objetivo, el objetivo que se persigue en la empresa es la minimización del makespan C_{max}. Mediante la reducción del makespan, se busca lograr un aprovechamiento mayor de las máquinas mediante la reducción de las esperas y las máquinas ociosas.

3. METAHEURÍSTICAS PROPUESTAS

"Industria es la aparente simpleza."
Séneca

EL problema definido como "two-stage assembly flowshop scheduling problem" ya ha sido estudiado con anterioridad con el objetivo de minimizar el makespan y sin ninguna restricción. Lee (1993) estudió el problema considerando $m = 2$ mientras que Potts (1995) lo hizó para m. Ambos llegaron a la conclusión de que no es factible la resolución del problema mediante el estudio de todas y cada una de las posibles soluciones, clasificando el problema en NP-hard. De esta forma, puesto que el problema para $m = 2$ $F2 \rightarrow 1|\ prmu\ |C_{\max}$ es NP-hard, para $Fm \rightarrow 1|s_{ij}, prmu\ , skip|C_{\max}$ podemos decir que también lo es, puesto que $m\ =\ 2$ es una particularización del caso genérico. Por lo tanto, nos serviremos de metaheurísticas para la resolución de este modelo. En este capítulo se va a realizar una breve descripción sobre las metaheurísticas existentes más famosas, siguiendo la clasificación: metaheurísticas basadas en trayectoria y metaheurísticas basadas en población.

3.1. Clasificación de las metaheurísticas

Existen numerosas formas de clasificar las diversas técnicas metaheurísticas. Según las características seleccionadas, se pueden obtener diferentes clasificaciones: basadas o no en la naturaleza, con o sin memoria… Una de las clasificaciones más populares, según explica Chicano (2007) en su tesis doctoral, es la que divide las metaheurísticas en basadas en trayectoria y basadas en población. Esta taxonomía puede observarse en la gráfica de la figura 8, en la que además se han incluido las principales metaheurísticas.

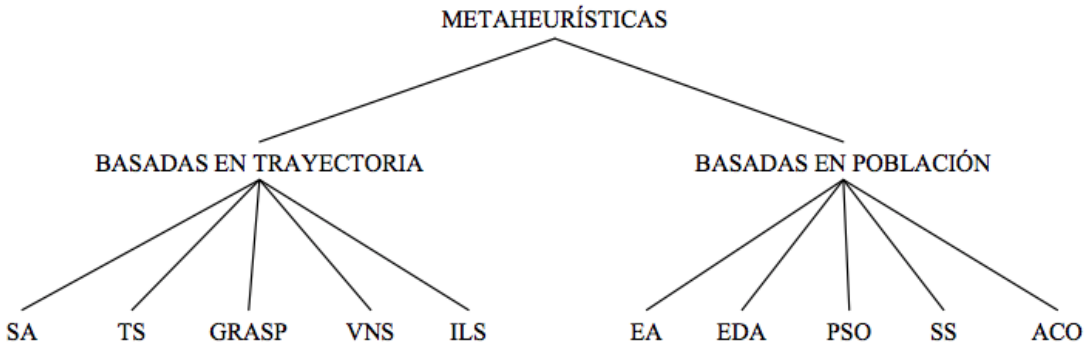

***Figura 8**: clasificación de metaheurísticas. Fuente: Tesis doctoral José Francisco Chicano [2007]*

3.1.1. Metaheurísticas basadas en trayectoria

Estos métodos se caracterizan por partir de una solución inicial y, mediante la exploración del vecindario, actualizar la solución actual, formando una trayectoria. La mayoría de estos procesos han surgido de añadir algún mecanismo de perturbación a las soluciones obtenidas de las búsquedas locales simples, lo que les permite escapar de los óptimos locales. Es necesario definir un criterio de parada, como detener la búsqueda tras un número de iteraciones, al encontrar una solución suficientemente buena o cuando se detecta un estancamiento.

Las metaheurísticas basadas en trayectoria más conocidas son:

- **Enfriamiento simulado (SA)**: este método trata de simular el proceso de enfriamiento del metal y del cristal. Con el fin de evitar caer en óptimos locales, el algoritmo permite seleccionar una solución cuyo valor de fitness[1] sea peor que el de la solución actual. En cada iteración se selecciona un vecino obtenido a partir de la solución actual. Si el vecino es mejor a la solución actual, esta es sustituida. En caso contrario, la posibilidad de aceptación depende de la temperatura actual T y de la diferencia de fitness de ambas soluciones.

- **Búsqueda tabú (TS):** este método se basa en la utilización de un historial de búsqueda (una memoria a corto plazo), para escapar de mínimos locales y para evitar la búsqueda del óptimo en la misma región. En la memoria a corto plazo se implementa una lista, conocida como lista tabú, en la cual se mantienen las ultimas soluciones visitadas para evitar seleccionarlas en las próximas iteraciones. La mejor solución obtenida en cada iteración es añadida a la lista tabú.

- **GRASP:** sus siglas equivalen a *Greedy Randomized Adaptive Search Procedure* o procedimiento de búsqueda miope aleatorizado y adaptativo. Esta metaheurística se compone de un proceso iterativo en dos fases: en primer lugar, la construcción de una solución y en segundo lugar la aplicación de un proceso de mejora a este algoritmo. El proceso de

[1] Fitness: equivale a la FO.

construcción de la solución es un heurístico constructivo aleatorio, en el que se van añadiendo paso a paso diferentes componentes a la solución parcial, que se encuentra vacía al comienzo del proceso. En la segunda fase se aplica un método de búsqueda local que puede ser simple o complejo como SA o TS.

- **Búsqueda con vecindario variable (VNS):** esta metaheurística se basa en el intercambio entre distintos vecindarios durante el proceso de búsqueda. Al comienzo del proceso, se debe definir un conjunto de vecindarios. Estos pueden ser seleccionados de manera aleatoria o mediante técnicas más complejas como el uso de ecuaciones deducidas del problema. En cada iteración se suceden tres fases: la selección del candidato, la fase de mejora y el movimiento. Tras seleccionar un vecino del vecindario, se procede a la aplicación de un proceso de búsqueda local. Una vez este finaliza, se compara la mejor solución obtenida con la original. Si mejora a la inicial, se convierte en solución actual; si no lo es se repite el proceso utilizando el siguiente vecindario.

- **Búsqueda local iterada (ILS):** esta metaheurística sigue un procedimiento simple pero muy efectivo. Se parte de una solución inicial, la cual puede ser obtenida aleatoriamente o mediante algún proceso más complejo, como la aplicación de una regla de despacho. A esta solución inicial, se le aplica un movimiento de perturbación/diversificación para llegar a otras regiones. A la solución obtenida, se le aplica búsqueda local, lo que nos permite obtener un nuevo óptimo. Este óptimo puede ser aceptado o no, ya que existen dos criterios de elección: seleccionar la mejor solución o seleccionar siempre la nueva solución, a pesar de que esta sea peor, ya que de esta manera se diversifica y se evita caer en óptimos locales.

3.1.2. Metaheurísticas basadas en población

Estos métodos se caracterizan por partir de un conjunto de soluciones iniciales en cada iteración, a diferencia de los métodos basados en trayectoria en los que solo se trabaja con una única solución por iteración.

Las metaheurísticas basadas en población más conocidas son las siguientes:

- **Algoritmos evolutivos (EA):** esta metaheurística está basada en la teoría de la selección natural. Mediante un proceso iterativo que trabaja sobre una población de soluciones se modifica la población actual, formando la nueva generación del algoritmo con un fitness mejorado. Un algoritmo evolutivo puede estructurarse en tres fases: selección, reproducción y reemplazo. En la fase de selección se seleccionan los individuos más aptos o con mayor fitness para que pasen a la fase de reproducción. Los individuos son modificados mediante operadores de mutación y cruce. Posteriormente, un criterio de aceptación decide si entran o no en la población y a qué individuo sustituyen. Este proceso se repite iterativamente hasta alcanzar un criterio de terminación.

- **Algoritmos de estimación de la distribución (EDA):** estos algoritmos se comportan de manera similar a los algoritmos evolutivos. Este tipo de algoritmo persigue la estimación de la

distribución de probabilidad de cada variable. No existen operadores de cruzamiento ni de mutación para la generación de nuevos individuos, sino que se obtienen desde las distribuciones de probabilidad.

- **Búsqueda dispersa (SS):** esta metaheurística mantiene un conjunto de soluciones llamado conjunto de referencia que se caracteriza por contener soluciones de alta calidad y dispersas en el espacio. Sus fases son cinco: creación de la población inicial, generación del conjunto de referencia, generación de soluciones, combinación de soluciones y método de mejora.

- **Optimización basada en colonias de hormigas (ACO):** estos algoritmos están basados en el comportamiento de hormigas cuando se encuentran en búsqueda de comida. Las hormigas exploran el área cercana a su hormiguero de manera aleatoria hasta que encuentra comida y la lleva al hormiguero. Mientras realiza todo este camino, la hormiga segrega feromona que ayudará al resto de la colonia a llegar hasta la comida. La técnica de optimización actúa de manera similar, construye una solución inicial aleatoria basada en el comportamiento de una hormiga y “deja rastros de feromonas” de manera artificial que ayuden a encontrar el camino más corto.

- **Optimización basada en cúmulos de partículas (PSO):** estos algoritmos están basados en el comportamiento de las bandadas de aves durante el vuelo o en los movimientos de los bancos de peses. El algoritmo inicializa un conjunto de soluciones, a las que también se les puede llamar partículas, de manera aleatoria. Cada partícula posee una posición y una velocidad que varía en cada iteración. En el movimiento de la partícula, además de la velocidad y posición, influyen los lugares donde esta y las partículas de su vecindario encontraron buenas soluciones.

4. ITERATED LOCAL SEARCH (ILS)

La calidad nunca es un accidente; siempre es el resultado de un esfuerzo de la inteligencia.
Jonh Ruskin

PARA la resolución del problema expuesto en el capítulo 2, consistente en un entorno *assembly* en dos fases con el objetivo de minimización del makespan, se utilizará la metaheurística conocida como *Iterated Local Search*. De entre todos los algoritmos expuestos en el capítulo 3, se ha elegido este debido a que según Juan. A. et. al. (2012) desde el punto de vista de la gestión, cuanto más eficiente y simple sea un algoritmo, más probable es su uso para solucionar problemas de la vida real. Esta metaheurística es la que mejor cumple con este principio. Además, el ILS es una de las metaheurísticas más usada para entornos flowshop, como el que se puede encontrar en la primera fase del entorno del problema. A pesar de que el algoritmo ha sido explicado anteriormente en la sección 3.1, hablaremos de él con más detalle.

La idea esencial del Iterated Local Search radica en enfocar la búsqueda en un pequeño subespacio de soluciones definido por las soluciones que son óptimos locales.

En primer lugar, se debe crear una solución inicial, la cuál además es la mejor solución hasta el momento. Esta solución puede generarse aleatoriamente, siguiendo alguna regla de despacho o mediante cualquier otro método que nos pueda proporcionar una solución inicial lo suficientemente buena.

Después se realiza el proceso de búsqueda de posibles soluciones que mejoren el valor de la FO. Esta búsqueda se realiza mediante la aplicación de algún tipo de movimiento en la vecindad. Existen tres tipos de vecindades:

- **Intercambio adyacente:** consiste en el intercambio de dos trabajos adyacentes en la secuencia de trabajos. Una secuencia S de n trabajos tendrá $n - 1$ vecinos.

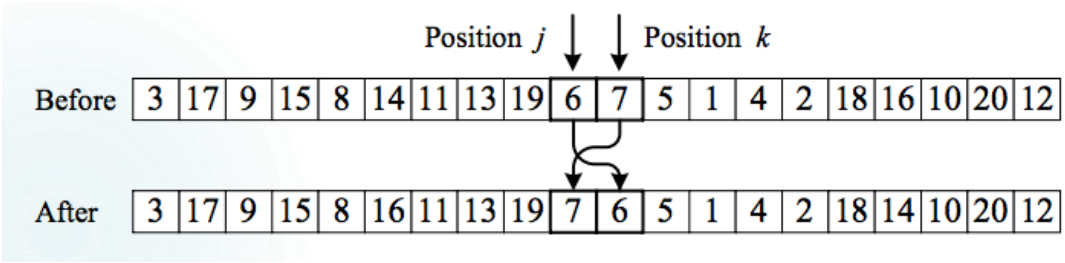

***Figura 9**: Intercambio adyacente. Fuente: Diapositivas Programación y Control de la Producción [2017].*

- **Intercambio:** consiste en el intercambio de dos trabajos sin que tengan que ser adyacentes. Una secuencia S de n trabajos tiene $\frac{n(n-1)}{2}$ vecinos.

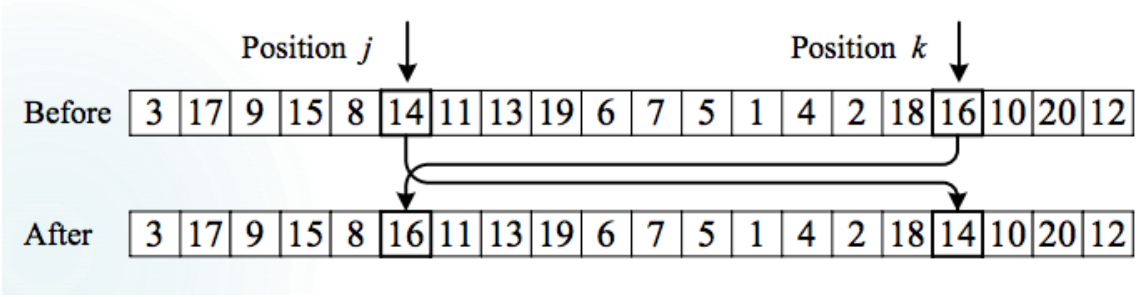

***Figura 10:** Intercambio. Fuente: Diapositivas Programación y Control de la Producción [2017].*

- **Inserción:** consiste en la extracción de un elemento de la secuencia S de n trabajos y en su inserción en todas las posiciones posibles de la secuencia.

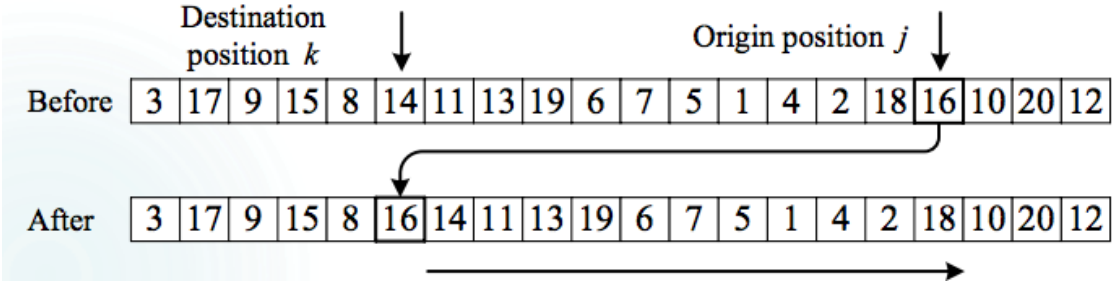

***Figura 11:** Inserción. Fuente: Diapositivas Programación y Control de la Producción [2017].*

El proceso de búsqueda puede realizarse según dos criterios:

- **Best improvement**: se analiza la vecindad entera, luego se compara el mejor de los vecinos con la mejor solución obtenida hasta el momento. En el caso de que esta solución sea mejor pasa a reemplazarla.

- **First improvement**: se analiza la vecindad hasta obtener un vecino cuya FO sea mejor a la mejor solución obtenida hasta el momento. El vecino pasa a ser la mejor solución.

Partiendo de la solución obtenida, se vuelve a realizar el proceso de búsqueda. Una vez se ha realizado un número prefijado de iteraciones sin encontrar ninguna mejora, se añade una perturbación a la mejor solución obtenida hasta el momento.

Para la solución partida a través de la perturbación, se vuelve a realizar el proceso de búsqueda en su vecindad. Deteniendo el proceso cuando no se encuentre ninguna solución que mejore el óptimo ya obtenido.

Todo este proceso, se realizará tanto para el problema que trata de solucionar este problema como para una batería de problemas con distinto número de trabajos, etapas y % de trabajos que no pasan por una determinada máquina. Se han desarrollado dos variaciones de este algoritmo para obtener una solución óptima al problema basándonos en el objetivo de minimización del makespan.

4.1. Algoritmo Iterated Local Search 1

Esta primera versión del algoritmo, partirá de una solución inicial que ha sido generada de manera aleatoria.

A esta solución inicial, se le realiza un estudio de su vecindad mediante inserción. Es decir, se extraen todos los trabajos de la secuencia uncial y se extraen en todas las posiciones de la secuencia. Cada una de estas combinaciones, se denomina vecino. Se evalúan los valores de la FO para todos los vecinos.

Una vez analizada toda la vecindad, es decir, todos los trabajos se han insertado en todas las posiciones. Se selecciona la mejor solución obtenida entre todos los vecinos. Posteriormente, se compara este mejor vecino con la mejor solución obtenida hasta el momento por el algoritmo.

En caso de que el mejor vecino mejore el valor de la mejor solución obtenida hasta el momento, se realiza un estudio de la vecindad de esta secuencia en la siguiente iteración.

En caso contrario, se añade una perturbación al mejor vecino obtenido en la iteración. Esta perturbación se realiza mediante intercambio. Se seleccionarán dos posiciones aleatorias de la secuencia y se intercambiarán sus posiciones.

Este proceso se continúa realizando hasta que se produzcan 10 iteraciones en las cuáles no se encuentre una solución en la cual se mejor la mejor solución obtenida hasta el momento.

4.2. Algoritmo Iterated Local Search 2

En esta segunda versión del algoritmo, se partirá de una solución inicial generada de manera aleatoria.

A esta solución inicial, se le realiza un estudio de su vecindad mediante intercambio adyacente. Es decir, se intercambian las posiciones de todos los trabajos con la de su trabajo adyacente. Se evalúan los valores de la FO para todos los $\frac{n(n-1)}{2}$ vecinos.

Una vez analizada toda la vecindad, es decir, todos los trabajos se han intercambiado con su adyacente correctamente. Se selecciona la mejor solución obtenida entre todos los vecinos. Posteriormente, se compara este mejor vecino con la mejor solución obtenida hasta el momento por el algoritmo.

En caso de que el mejor vecino mejore el valor de la mejor solución obtenida hasta el momento, se realiza un estudio de la vecindad de esta secuencia en la siguiente iteración.

En caso contrario, se añade una perturbación al mejor vecino obtenido en la iteración. Esta perturbación se realiza mediante intercambio. Se seleccionarán dos posiciones aleatorias de la secuencia y se intercambiarán sus posiciones.

Este proceso se continúa realizando hasta que se produzcan 10 iteraciones en las cuáles no se encuentre una solución en la cual se mejor la mejor solución obtenida hasta el momento.

5. IMPLEMENTACIÓN

"La producción de demasiadas cosas útiles da lugar a demasiadas personas inútiles."
Karl Max

En este capítulo vamos a describir la implementación de los algoritmos explicados en el capítulo 4 para la resolución del problema de la producción citado anteriormente (capítulo 2). Para su implementación, se ha optado por la programación del algoritmo usando el lenguaje C debido a la flexibilidad del lenguaje. Los códigos íntegros pueden encontrarse en el Anexo II, en este capítulo se procederá a explicar pequeñas partes de estos.

Los códigos han sido desarrollados con la ayuda del compilador Code::Blocks, entorno de desarrollo libre (GPL: *General Public License*), que permite el desarrollo de aplicaciones en lenguaje C y C++. En el desarrollo del código se ha hecho uso de la librería "Schedule" aportada por el personal docente de la asignatura "Programación y Control de la Producción" del Grado en Ingeniería de Organización Industrial.

La codificación del algoritmo ILS explicado en el capítulo 4 tiene por objetivo encontrar la mejor secuencia posible para el problema, además de devolvernos el valor de su FO, en este caso C_{max}. Se realizarán dos variantes del algoritmo. Después, se estudiará cuál es la que aporta mejor solución al problema.

Además, para poder comparar correctamente ambas variantes del algoritmo, se aplicarán a una batería de problemas con distinto número de trabajos, etapas y % de operaciones que se saltarán las etapas (o % missing). El objetivo de esto es comprobar la eficiencia de estos algoritmos en entornos similares al estudiado en este trabajo.

Una vez obtenidas las soluciones por medio del código generado, se exportarán a la aplicación Microsoft Excel.

5.1. Versión 1 del Algoritmo ILS

Se realizarán dos versiones para resolver cada uno de los problemas generados anteriormente de manera aleatoria.

En este primer algoritmo, se realizará una función que estudie la vecindad de la solución inicial mediante inserción. Una vez se hayan insertado todos los trabajos en todas las posiciones, se comparará la mejor solución obtenida en esta iteración con la mejor solución obtenida hasta el momento. En caso de que la solución obtenida en la iteración sea mejor a la mejor solución obtenida hasta el momento, se volverá a estudiar la vecindad partiendo de esta secuencia. En caso contrario, se perturbará la secuencia intercambiando dos posiciones aleatorias. Se volverá a realizar la vecindad para esta secuencia perturbada. El proceso se repetirá hasta que se realicen 10 iteraciones en las cuáles no se encuentre una solución que mejore al valor de la mejor FO obtenida hasta el momento. Es decir, este algoritmo utiliza una vecindad por inserción con *best improvement* para la resolución del problema, realizando una perturbación por intercambio en caso de no encontrar una solución mejor.

Todo este proceso, se realizará en C mediante una función. La función recibirá como parámetros los tiempos de proceso de la fase 1 y de la fase de *assembly*, los tiempos de trabajo, el número de trabajos y el número de máquinas. La función devuelve el valor de la mejor FO obtenida.

```
int IteratedLocalSearch1 (MAT_INT p_ij, VECTOR_INT seqOr, int n, int
m, int p_single)
```

En primer lugar, se procede a la definición e inicialización de todos los vectores y variables auxiliares necesarias para el algoritmo.

Para el estudio de la vecindad, son necesarios dos bucles for. En el primero de ellos, se selecciona la posición del trabajo a insertar en todas las posiciones del vector secuencia y extrayéndolo de esta. En el segundo bucle for, se recorrerán todas las posiciones del vector, insertando el trabajo extraído en todas ellas.

```
for(i=0; i<n; i++)
        {
         int a = seq[i];
         extractIVector(seq_iter, n, i);
         copyIVector(seq_iter, vecinos, n-1);
         for(j=0; j<n; j++)
         {
              insertIVector(vecinos, n, a, j);
              FO_vecino = FO_Cmax(p_ij, n, m, vecinos, p_single);
```

Para cada vecino obtenido, se calcula el valor de su FO objetivo. Posteriormente, se comparará su valor con el mejor obtenido en la iteración hasta el momento. En caso de ser mejor, se almacena como la mejor vecindad de la iteración.

Una vez estudiada toda la vecindad, se compara la mejor solución obtenida en la iteración (en caso de no haberla se compara la secuencia inicial de la vecindad) con la mejor solución obtenida en el algoritmo. En caso de que la solución obtenida de la vecindad sea mejor, se vuelve a realizar la vecindad partiendo de esta secuencia. En caso contrario, se procede a perturbar la secuencia inicial de la vecindad.

Esta perturbación, se realiza mediante intercambio, seleccionando dos posiciones aleatorias del vector e intercambiando sus valores.

```
int posAleatoria2,aux2;
            int posAleatoria1;
            posAleatoria1 = rand()%n;
            do
            {
                posAleatoria2 = rand()%n;
            }
            while(posAleatoria1==posAleatoria2);

            aux2 = seq_iter[posAleatoria1];
            seq_iter[posAleatoria1] = seq_iter[posAleatoria2];
            seq_iter[posAleatoria2] = aux2;
```

Para evitar que se seleccionen las mismas dos posiciones aleatorias para el intercambio, se emplea un bucle do-while. En caso de que la segunda posición aleatoria coincida con la posición aleatoria obtenida en primer lugar, se ejecutara el bucle hasta obtener una posición distinta.

Este proceso se repite hasta que se suceden 10 iteraciones en las cuáles no se ha obtenido ninguna mejora.

Finalmente, se procede a guardar en un fichero el número de iteraciones que han sido necesarias hasta llegar a la solución obtenida.

```
FILE *resultadosalgoritmo1;
resultadosalgoritmo1=fopen("Resultadosalgoritmo1.txt", "a");
fprintf(resultadosalgoritmo1, "\nIteraciones algoritmo 1: %d ",
iter);
fclose(resultadosalgoritmo1);
```

5.2. Versión 2 del Algoritmo ILS

En este segundo algoritmo, se realizará una función que estudie la vecindad de la solución inicial mediante intercambio adyacente. Una vez se hayan intercambiado todos los trabajos con sus adyacentes, se comparará la mejor solución obtenida en esta iteración con la mejor solución obtenida hasta el momento. En caso de que la solución obtenida en la iteración sea mejor a la

mejor solución obtenida hasta el momento, se volverá a estudiar la vecindad partiendo de esta secuencia. En caso contrario, se perturbará la secuencia intercambiando dos posiciones aleatorias. Se volverá a realizar la vecindad para esta secuencia perturbada. El proceso se repetirá hasta que se realicen 10 iteraciones en las cuáles no se encuentre una solución que mejore al valor de la mejor FO obtenida hasta el momento. Es decir, este algoritmo utiliza una vecindad por intercambio adyacente con *best improvement* para la resolución del problema, realizando una perturbación por intercambio en caso de no encontrar una solución mejor.

Todo este proceso, se realizará en C mediante una función. La función recibirá como parámetros los tiempos de proceso de la fase 1 y de la fase de *assembly*, el número de tiempos de trabajo y de máquinas. La función devuelve el valor de la mejor FO obtenida hasta el momento.

```
int IteratedLocalSearch2 (MAT_INT p_ij, VECTOR_INT seqOrigen, int n,
int m, int p_single)
```

En primer lugar, se procede a la definición e inicialización de todos los vectores y variables auxiliares necesarias para el algoritmo.

Para la realización del intercambio adyacente en la vecindad, es necesario un único bucle for. Este bucle recorrerá todas las posiciones del vector secuencia, intercambiándolas por el elemento que se encuentra en su posición consecutiva.

Una vez realizado el intercambio, se calcula el valor de su FO. En caso de que esta solución tenga un valor inferior al mejor que se ha obtenido hasta el momento, se almacenan como los óptimos hasta el momento.

Después de esto, se deshace el cambio y se procede a realizar el intercambio adyacente del siguiente elemento del vector secuencia.

```
for(i=0; i<n-1; i++)
  {
       aux = seq[i];
       seq[i]= seq[i+1];
       seq[i+1]=aux;

       FOseq = FO_Cmax(p_ij, n, m, seq, p_single);

       if(FOseq<FO_opt)
       {
           mejorado = 1;
           FO_opt = FOseq;
           copyIVector(seq, seq_opt, n);
        }
        aux = seq[i];
        seq[i]= seq[i+1];
        seq[i+1]=aux;
```

```
}
```

En caso de que se haya encontrado una solución que mejore la mejor solución obtenida hasta el momento, se vuelve a realizar el intercambio adyacente con la mejor solución obtenida. En caso contrario, se procede a perturbar la secuencia inicial de la vecindad.

```
int posAleatoria2,aux2;
int posAleatoria1 = rand()%n;
do
{
        posAleatoria2 = rand()%n;
}
while(posAleatoria1=posAleatoria2);
aux2 = seq[posAleatoria1];
seq[posAleatoria2] = seq[posAleatoria1];
seq[posAleatoria1] = aux2;
```

Para evitar que se seleccionen las mismas dos posiciones aleatorias para el intercambio, se emplea un bucle do-while. En caso de que la segunda posición aleatoria coincida con la posición aleatoria obtenida en primer lugar, se ejecutara el bucle hasta obtener una posición distinta.

Este proceso se repite hasta que se suceden 10 iteraciones en las cuáles no se ha obtenido ninguna mejora.

Finalmente, se procede a guardar en un fichero el número de iteraciones que han sido necesarias hasta llegar a la solución obtenida.

```
FILE *resultadosalgoritmo2;
resultadosalgoritmo2 = fopen("Resultadosalgoritmo2.txt", "a");
fprintf(resultadosalgoritmo2, "\nIteraciones algoritmo 2: %d ",
iter);
fclose(resultadosalgoritmo2);
```

5.3. Resolución de las instancias

La implementación de los algoritmos a las diferentes instancias podría dividirse en dos etapas. La primera de ellas, sería en la cual se procede a la generación de la batería aleatoria de problemas, y a la resolución de esta mediante ambos algoritmos. Una segunda fase posterior, que puede verse en la sección 5.3.2 será la implementación de ambos algoritmos para la resolución del problema en la empresa, con sus datos reales.

5.3.1. Generación de batería de problemas

En primer lugar, se procederá a la creación de una batería aleatoria de problemas que nos permitirán comparar la eficiencia de ambos algoritmos. Las variables a tener en cuenta en la generación de problemas serán el porcentaje de missing, el número de trabajos y el número de etapas.

La generación de esta batería de problemas dentro del código elaborado, se realiza por medio de una función. Cada uno de los problemas a resolver tienen una serie de parámetros asociados:

- número de trabajos (n)
- número de máquinas (m)
- tiempos de proceso en la primera fase, antes del assembly (P_{ij})
- tiempos de proceso en la fase de assembly (P_i)
- secuencia inicial (seq)
- valor de la FO (FO)

Por ello, debido a la necesidad de agrupar datos que serían muy difícil de manejar de manera individual se ha recurrido a la creación de una estructura para su almacenamiento. La definición de la estructura se ha realizado al principio del código del programa, con el código:

```
struct instancia
{
	int n;
	int m;
	int ps;
	float P;
	long int FO;
	MAT_INT Pij;
	VECTOR_INT seq;
};
```

Posteriormente, se ha realizado la siguiente función en la cual todos estos datos (menos el valor de la FO) se generan de manera aleatoria. La función recibe como parámetro de entrada un vector de estructuras de longitud 360, es decir, el número total de problemas que se van a estudiar.

```
void generarInstancia(struct instancia instancias[360])
```

Posteriormente, se procede a crear las 360 instancias para el problema. Los problemas, deben responder a unos determinados parámetros. Para cada combinación de estos parámetros, se crean un total de 10 instancias, obteniendo finalmente una batería de 360 problemas a resolver. La información más detallada acerca de los valores de los parámetros usados se encuenra en el capítulo 6.

Para que cada una de las instancias generadas responda a estos valores, en primer lugar, se procede a la creación de tres vectores en los que se almacenan los valores posibles de estos parámetros.

```
int M[3] = {7,14,28};
int N[3] = {4,8,16};
float MS[4] = {0,0.25,0.50,0.60};
```

Posteriormente, para asignar valores a cada una de estas instancias, se recurrirá a un árbol de bucles donde se incluirá un bucle por cada elemento. El primer bucle, será el correspondiente al número de máquinas. El segundo bucle, será el relacionado con el número de trabajos. El tercer bucle se relaciona con el porcentaje de missing. El cuarto bucle será el correspondiente a la creación de las 10 instancias para cada combinación de problemas.

Dentro de este árbol de bucles, se da valor al número de máquinas, número de trabajos y porcentaje de missing de acuerdo a los recogidos en los vectores indicados anteriormente. También se asigna un valor aleatorio al tiempo de proceso en la etapa de assembly, tiempos de proceso en la primera fase y una secuencia inicial aleatoria.

```
for(i=0; i<3; i++)
{
  for(j=0; j<3; j++)
  {
     for(k=0; k<4; k++)
     {
        for(l=0; l<10; l++)
        {
           instancias[in].m=M[i];
           instancias[in].n=N[j];
           instancias[in].P=MS[k];
           instancias[in].ps=rand()%99+1;
           randSequence(instancias[in].seq, instancias[in].n);
           for(trabajos=0; trabajos<instancias[in].n; trabajos++)
             {
               for(maquinas=0; maquinas<instancias[in].m; maquinas++)
                {
                    instancias[in].Pij[maquinas][trabajos] =
                    rand()%99+1;
```

Sin embargo, los tiempos de proceso generados aleatoriamente en este árbol de bucles no satisfacen el parámetro correspondiente al % de missing. Para ello, se genera el siguiente trozo de código:

```
ponerCero = (int)( instancias[in].P
*(instancias[in].m*instancias[in].n));
       do
```

```
        {
            posX=rand()%instancias[in].m;
            posY=rand()%instancias[in].n;
            if(instancias[in].Pij[posX][posY]>0 &&ponerCero>0)
            {
                  instancias[in].Pij[posX][posY]=0;
                  ponerCero--;
             }
         }
         while(ponerCero>0);
```

En primer lugar, se calculan el número de trabajos que no deben pasar por ninguna máquina que deben existir en la instancia. Posteriormente, se emplea un bucle do-while para sustituir posiciones aleatorias de la matriz de tiempos de proceso por ceros, siempre y cuando no haya un cero de manera previa en esa posición. El bucle se repite hasta que se cumple el % de missing en la matriz.

Una vez definida la función para la generación de la batería aleatoria de 360 problemas, se debe realizar una llamada a ella en la función main.

Cada una de las funciones correspondientes a los algoritmos, devuelve un valor de la FO. Para el almacenamiento de los valores de las distintas soluciones, se crearán dos vectores con dimensión igual al número de instancias, 360.

```
generarInstancia(instancias_main);

VECTOR_INT res_algoritmo1 = DIM_VECTOR_INT(360);
VECTOR_INT res_algoritmo2 = DIM_VECTOR_INT(360);
```

Para la resolución de las 360 instancias almacenadas en el vector de estructuras, se emplea un bucle for. Dentro de este bucle, se realiza una llamada a las funciones correspondientes a los dos algoritmos comentados en los apartados 5.1 y 5.2. Los valores devueltos por estas funciones, serán almacenados en los vectores comentados anteriormente.

```
for(i=0; i<360; i++)
 {
      res_algoritmo1[i] = IteratedLocalSearch1(instancias_main[i].Pij,
      instancias_main[i].seq, instancias_main[i].n, instancias_main[i].m,
      instancias_main[i].ps);
      res_algoritmo2[i] = IteratedLocalSearch2(instancias_main[i].Pij,
      instancias_main[i].seq, instancias_main[i].n, instancias_main[i].m,
      instancias_main[i].ps)
 }
```

Finalmente, se almacenarán estos datos en un fichero, con el fin de permitir su exportación a un archivo Excel para su posterior evaluación computacional en el capítulo 6.

```
FILE *resultados;
    resultados=fopen("Resultados.txt", "a");
    for(i=0; i<360; i++)
    {
      fprintf(resultados, "\n %d \t %d",res_algoritmo1[i],
      res_algoritmo2[i]);
    }
    fclose(resultados);
```

5.3.2. Resolución del problema inicial

Una vez realizado el código de ambos algoritmos, se debe realizar la implementación de ambos para que puedan resolver el problema inicial de la empresa real.

En primer lugar, se debe realizar la carga de los tiempos de procesos previamente copiados en un fichero de texto.

Al tener un entorno assembly, donde pueden diferenciarse dos fases, se emplearán dos ficheros, cada uno de ellos para una fase.

En la fase 1, además, existen tiempos de *setup* para algunos de los trabajos en algunas máquinas. Como estos tiempos de *setup* son independientes de la secuencia, y siempre se realizan una vez la pieza ha llegado a la máquina en la que debe ser procesada, se han sumado a los tiempos de proceso. De esta manera, se puede trabajar de manera más cómoda con una única matriz.

En la fase 2, tenemos un entorno *single machine*. Por lo tanto, tendremos un único tiempo de proceso, correspondiente a cuando los cuatro trabajos han sido procesados completamente en la fase 1. Al usar la función de la librería *Schedule* para cargar los tiempos de proceso, estos se almacenan en una matriz. Con el fin de poder trabajar más fácilmente con los datos, estos son exportados a una única variable entera.

```
MAT_INT p_ij_flow; //Cargamos tiempos de proceso de fase flowshop
p_ij_flow = loadPTimes_nrows("Tiemposdeproceso.txt",&n1, &m1, YES);
```

```
MAT_INT p_ij_single;
p_ij_single = loadPTimes_nrows("tiempossinglemachine.txt", &n2, &m2, YES);

int p_single;
p_single = p_ij_single[0][0];
```

Una vez se ha realizado satisfactoriamente la carga de los tiempos de proceso, se procede a definir una secuencia inicial aleatoria para el problema, mediante la función *randSequence*.

```
VECTOR_INT seq = DIM_VECTOR_INT(n1);
randSequence(seq,n1);
```

Posteriormente, calcularemos el valor de la FO en esta secuencia inicial, para que posteriormente podamos comparar como mejora cada uno de los algoritmos el valor C_{max} de la secuencia inicial.

```
FO_inicial = FO_Cmax(p_ij_flow, n1, m1, seq, p_single);
printf("El valor de la FO de la secuencia inicial es: %d \n", FO_inicial);
```

Finalmente, se llama a las funciones correspondientes a cada uno de los algoritmos que nos darán como salida el valor C_{max} por cada uno de los métodos. Tal y como se ha explicado en los apartados anteriores dedicados a ambos algoritmos, cada uno de estos algoritmos almacenará en un fichero el número de iteraciones necesarias hasta llegar a su solución obtenida.

```
FO_alg1 = IteratedLocalSearch1(p_ij_flow, seq, n1, m1, p_single);
FO_alg2 = IteratedLocalSearch2(p_ij_flow, seq, n1, m1, p_single);
```

6. RESULTADOS COMPUTACIONALES

"El experimentador que no sabe lo que está buscando no comprenderá lo que encuentra."
Claude Bernard

EN este capítulo se proceden a analizar los datos obtenidos de los códigos implementados en el Capítulo 5 para su correcta comprensión. Se procederá a analizar los dos algoritmos implementados en función del valor de la FO para cada instancia y del número de iteraciones necesarias hasta llegar a ella según los parámetros: número de trabajos, número de máquinas, % de missing y tiempo empleado hasta llegar a la solución.

Se establecerán comparaciones entre ambos algoritmos de manera que podamos identificar diferencias entre estos y los escenarios en los cuáles dan mejores soluciones. Es decir, se realizará un análisis exhaustivo de los resultados obtenidos en la fase de implementación del código combinando todos los parámetros con las soluciones obtenidas para obtener conclusiones acertadas.

Para cada una de los valores que se han extraído de cada iteración (FO e iteraciones) se realizarán tres gráficas, cada una de las cuáles correspondiente a cada uno de los parámetros de cada problema.

Además, para el estudio de la FO se recurrirá a la utilización del factor ARPD que nos permitirá cuantificar cuán mejor es la solución de un algoritmo respecto a la otra. Este se calcula mediante la siguiente formula:

$$ARPD = \frac{FO - FO_{best}}{FO_{best}}$$

6.1. Evaluación computacional en batería de problemas

6.1.1. Generación de problemas

La primera fase de los experimentos consistiría en el testeo de ambos algoritmos implementados en el capítulo 5, secciones 5.1 y 5.2. Este testeo se realizará de manera previa a la aplicación de ambos algoritmos para la resolución del problema real con el fin de identificar en que escenarios proporcionan soluciones óptimas cada uno de ellos.

Este testeo se realizará sobre una batería aleatoria de problemas. Los parámetros que se tuvieron en cuenta para la generación de estos problemas son: el número de trabajos a procesar, el número de máquinas y el porcentaje de missing.

El porcentaje de missing considerado es: 0%, 25% y 50%. Los números de trabajo considerados son: 4, 8, 16 y 32. El número de etapas considerado es: F7$\rightarrow$1, F14$\rightarrow$1 y F28$\rightarrow$1.

Para cada una de estas combinaciones, se generaron 10 instancias, obteniendo un total de 360 problemas.

Para cada instancia, se han generado de manera aleatoria tiempos de procesos y secuencias iniciales de los trabajos.

En la tabla 4, pueden observarse un ejemplo de los problemas generados, acompañada del valor de la FO de la secuencia inicial, el valor de la FO obtenida con cada algoritmo, el valor del factor ARPD y el nº de iteraciones que han sido necesarias hasta llegar a esta solución. También se incluye el tiempo necesario para llegar hasta cada solución. Debido al gran volumen de datos obtenido de la resolución de las 360 instancias, en la tabla 4 se muestra un promedio de los valores obtenidos en las 10 instancias generadas para cada combinación de problemas.

			Algoritmo 1			Algoritmo 2		
M	N	%	ARPD	nº iter	ms	ARPD	nº iter	ms
7→1	4	0%	2,89%	12,9	0	1,09%	11,2	0
		25%	2,52%	11,4	0	0,00%	11,2	0
		50%	4,38%	12,9	0	0,15%	10,7	0
		60%	2,80%	12,5	0	1,46%	11	0
	8	0%	1,32%	15,1	0	0,48%	12,1	0
		25%	1,23%	16,3	0	0,73%	12,8	0
		50%	1,81%	15,2	0	1,26%	11,6	0
		60%	2,75%	14,6	0	0,98%	11	0
	16	0%	1,27%	20,3	8,1	0,15%	14,4	0
		25%	1,10%	18,7	3,2	2,14%	15,4	1,5
		50%	0,65%	20,2	4,8	2,72%	12,6	0
		60%	1,54%	18,5	3,2	1,75%	13	0
14→1	4	0%	1,21%	12,6	0	1,09%	10,8	0
		25%	3,29%	12,7	0	0,09%	11,5	0
		50%	3,90%	13,2	1,5	0,39%	11,6	0
		60%	2,70%	12,1	1,7	0,41%	11,3	0
	8	0%	1,14%	14,7	0,4	1,10%	10,8	0
		25%	1,70%	16,3	1,6	1,48%	14,3	0
		50%	1,52%	13,9	1,6	1,03%	11,7	0
		60%	1,10%	14	0	1,42%	12	0
	16	0%	1,14%	19,7	6,2	1,18%	13,9	0
		25%	0,11%	19,1	12,6	1,98%	13,9	0
		50%	0,63%	18,8	7,4	2,78%	14,9	0
		60%	1,02%	20,5	15,7	1,89%	13,2	0
28→1	4	0%	1,92%	12,8	0	0,15%	11,2	0
		25%	1,82%	13,7	0	0,35%	10,8	0
		50%	1,12%	12	0	0,83%	11,5	0
		60%	1,09%	12,9	0	0,84%	11	0
	8	0%	0,57%	14,7	1,5	2,20%	12,6	0
		25%	1,84%	15,2	1,9	1,04%	12,2	0
		50%	1,25%	15,2	1,6	3,17%	13,2	0
		60%	2,02%	15,1	1,6	0,97%	10,9	0
	16	0%	0,71%	19,5	10,7	0,45%	13,7	3,1
		25%	0,27%	21,5	21,2	1,26%	13,6	0
		50%	1,77%	17,7	11,2	0,27%	12,6	0,1
		60%	0,83%	20,3	14,3	1,61%	12,9	0

Tabla 4: *Problemas generados*

6.1.2. Análisis de resultados en función del tiempo

Se va a analizar el tiempo empleado en mili segundos por cada algoritmo en alcanzar su mejor solución. Puede comprobarse que gran parte de estos datos tienen el valor 0. Esto se debe a que la precisión de la función usada durante la implementación no capta estos valores por ser demasiado pequeños. Por tanto, todos los valores que aparecen con 0 es debido a que se ejecutan tan rápido que el programa les asigna el valor 0.

En promedio, el algoritmo 1 tarda 3,66 ms en analizar cada una de las instancias. El algoritmo 2, tarda tan solo 0,13 ms. Puede verse la gran diferencia de valores entre estos, pudiendo destacarse la gran eficiencia del algoritmo 2 frente al 1.

A continuación, se presentan tres gráficas sobre cómo influyen los distintos valores de cada uno de los parámetros en el tiempo que emplea cada algoritmo en alcanzar una solución.

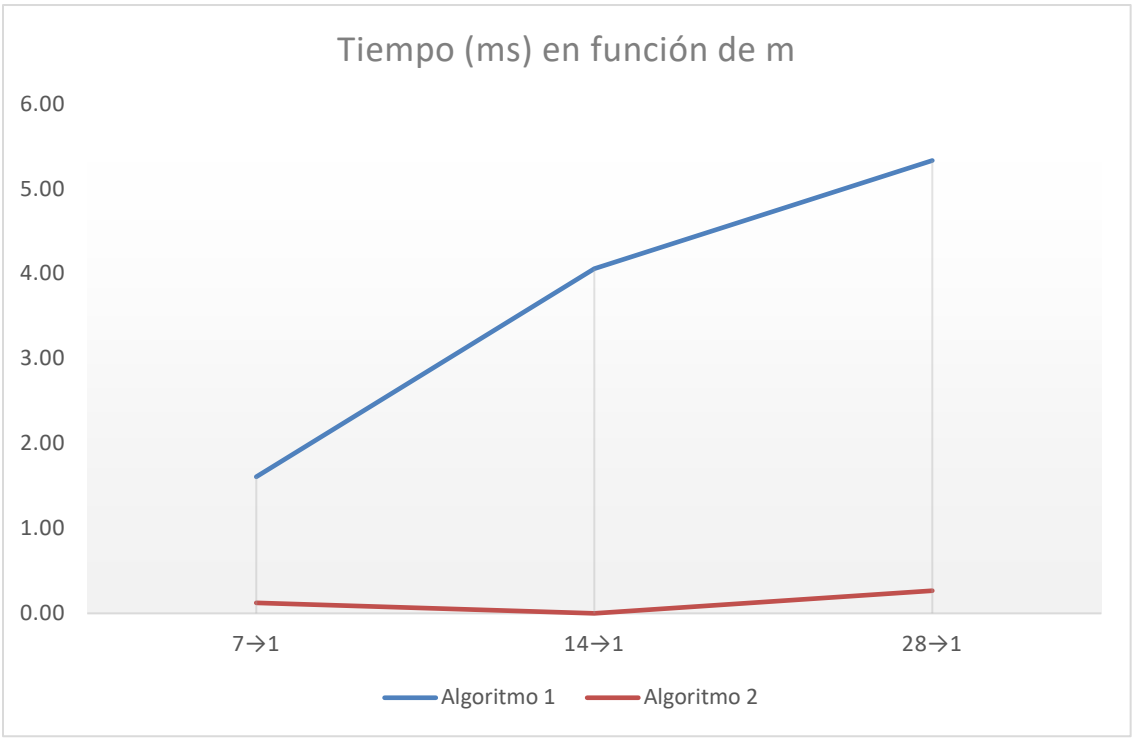

Figura 12: *Tiempo en ms en función del número de máquinas*

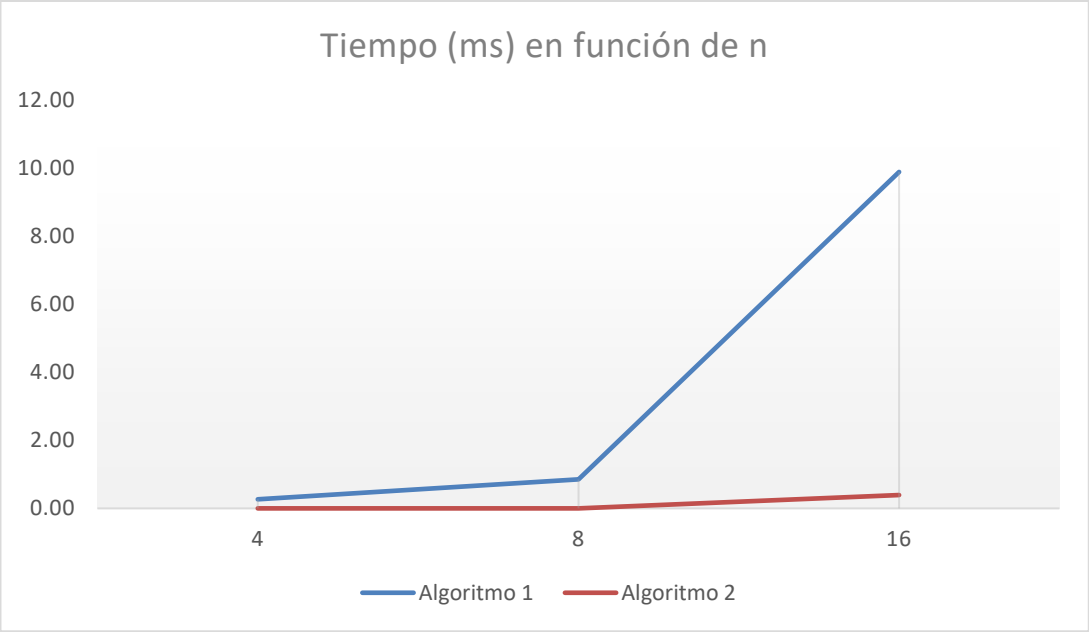

Figura 13: *Tiempo en ms en función del número de trabajos*

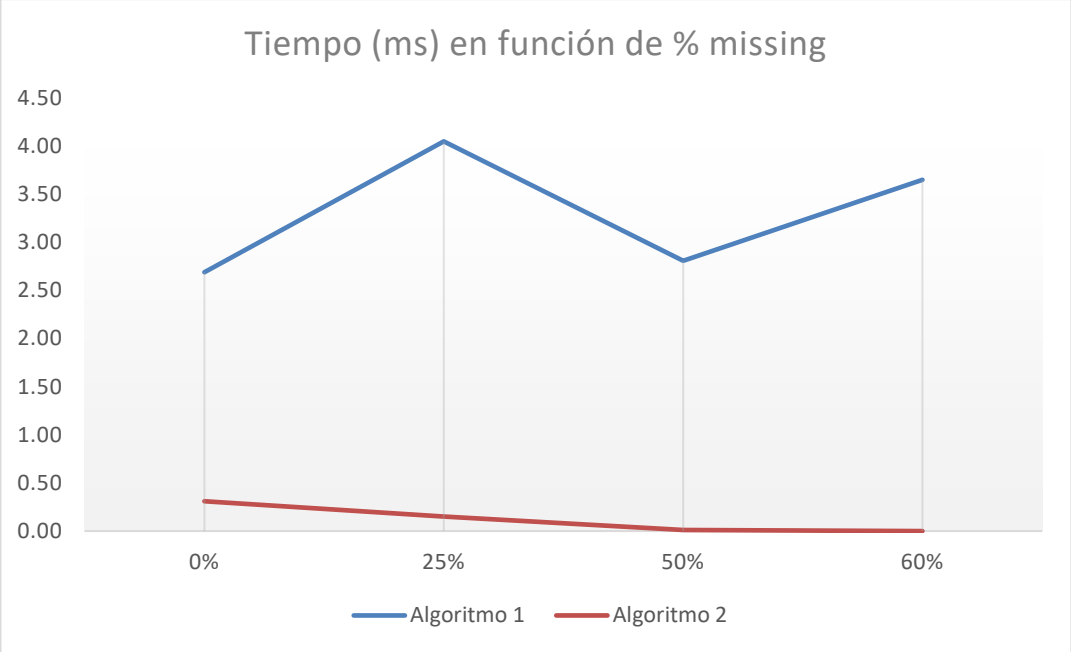

Figura 13: *Tiempo en ms en función del % missing*

Puede observarse como el tiempo empleado por el algoritmo 2 en encontrar una solución permanece prácticamente invariable frente a las variaciones de los distintos parámetros. Siendo además el tiempo de proceso bastante reducido

El algoritmo 1, sin embargo, presenta incrementos significativos en el tiempo empleado para encontrar una solución a medida que aumentan el número de trabajos y de máquinas. Frente a variaciones del parámetro % de missing no presenta ninguna tendencia tanto al alta como a la baja

6.1.3. Análisis de resultados en función del número de iteraciones

Se analiza el número de iteraciones necesarias hasta obtener la secuencia óptima en cada algoritmo para el problema.

En la tabla 5 se puede observar el número de iteraciones que emplea cada algoritmo de media hasta alcanzar su solución óptima.

	Algoritmo 1	Algoritmo 2
Media	15,74	12,30

Tabla 5: *Número medio de iteraciones hasta alcanzar una solución.*

A continuación, se presentan tres gráficas sobre cómo influyen los distintos valores de cada uno de los parámetros en el número de iteraciones.

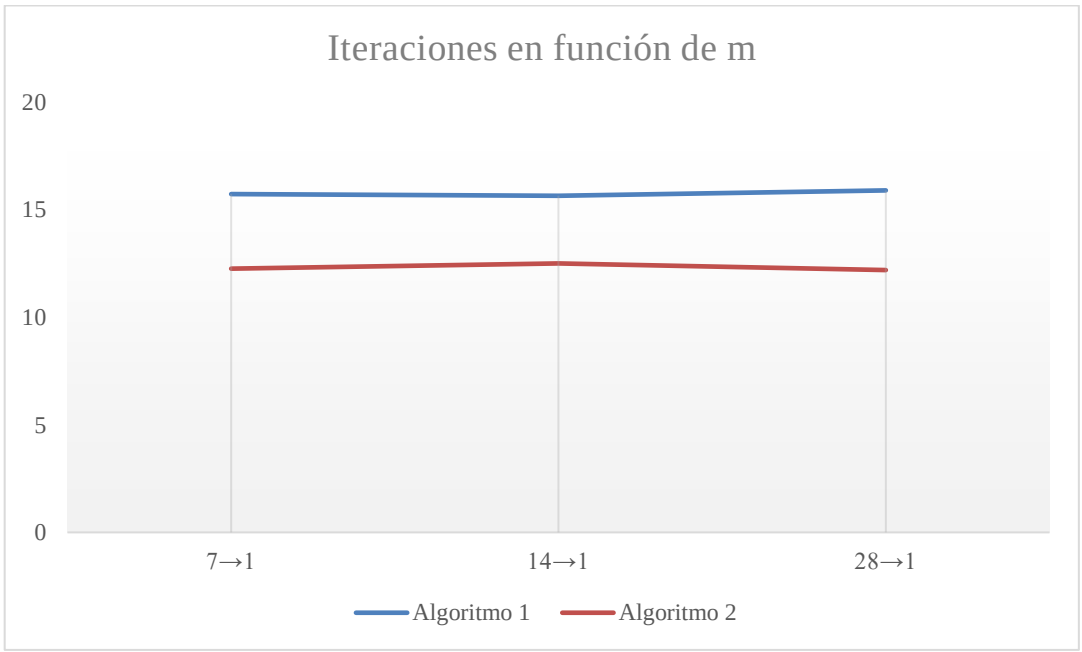

Figura 12: *Número de iteraciones en función del número de máquinas*

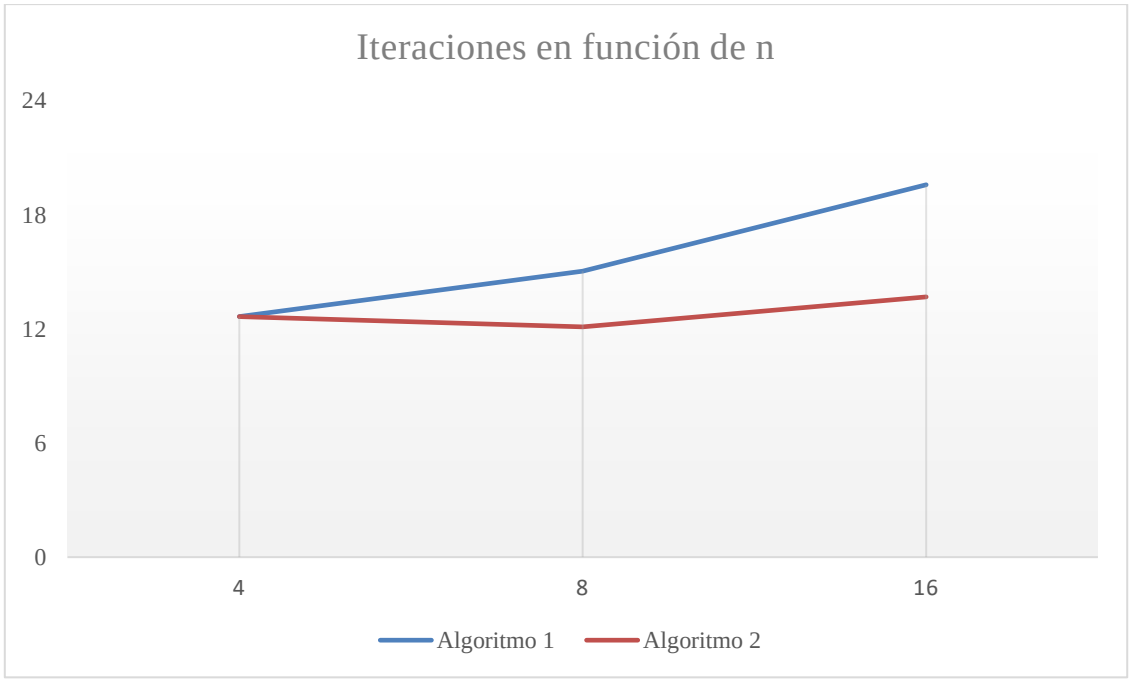

Figura 13: *Número de iteraciones en función del número de trabajos*

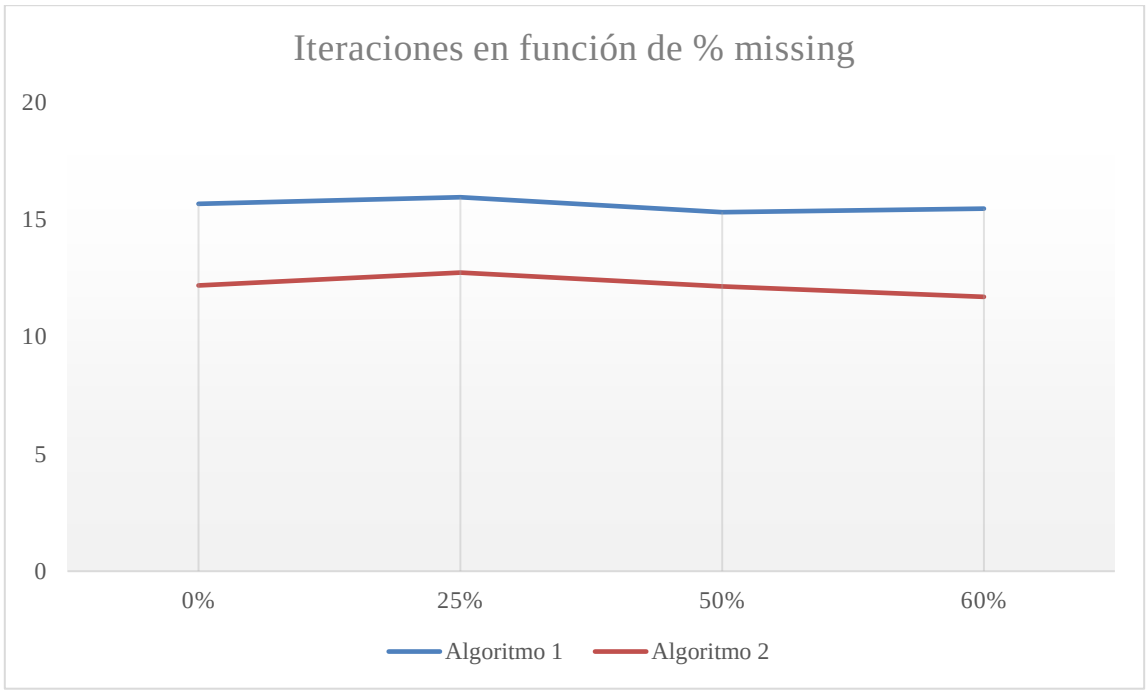

Figura 14: *Número de iteraciones en función del número del % de missing*

En las gráficas se puede observar claramente como el algoritmo 2 obtiene soluciones mucho más eficientes que el algoritmo 1, debido a que es necesario un menor número de iteraciones hasta llegar a ellas.

De media, los valores de los parámetros número de máquinas y % de missing no influyen en el número de iteraciones necesarias para llegar hasta la solución considerada por cada uno de los algoritmos.

Por lo tanto, se podría extraer como conclusión que, para el conjunto de parámetros estudiados, número de máquinas y % de missing, donde sus valores se encuentran acotados y con una variación relativamente pequeña entre los valores que estos pueden adoptar, ambos algoritmos necesitan de media el mismo número de iteraciones hasta llegar a una solución.

Sin embargo, en la figura 13, se puede observar como el número de iteraciones necesarias para que el algoritmo 1 alcance una solución crece de manera constante a medida que aumentan el número de trabajos. Por otra parte, se puede observar como el número de iteraciones necesarias en el algoritmo 2 se mantiene constante.

El algoritmo 1, responde de manera invariable ante las posibles variaciones en el número de máquinas y % de missing. Las variaciones en el número medio de iteraciones en cada uno de estos dos casos son despreciables, por lo que se podría decir que este algoritmo necesita el mismo número de iteraciones para resolver un problema, independientemente del número de trabajos y % de missing.

Sin embargo, este algoritmo presenta variaciones en el número de iteraciones necesarias en función al número de trabajos del problema, creciendo de manera lineal a medida que aumentan. Por tanto, cuanto mayor sea el número de trabajos a procesar en el problema, mayor será el número de iteraciones necesarias, por tanto, mayor tiempo hasta obtener la solución, por lo que este algoritmo es menos eficiente que el algoritmo 2.

De los datos contenidos en el Anexo 1, se puede extraer que este algoritmo presenta una variación del 61,53% en el número de iteraciones necesarias hasta alcanzar una solución.

El algoritmo 2, responde de igual manera antes las posibles variaciones en el número de trabajos, número de máquinas y % de missing. Las variaciones en el número medio de iteraciones en cada uno de los casos son despreciables, por lo que se podría decir que este algoritmo necesita el mismo número de iteraciones para resolver un problema, independientemente de sus parámetros.

De los datos contenidos en el Anexo 1, se puede extraer que este algoritmo presenta una variación del 56,52% en el número de iteraciones necesarias.

6.1.4. Análisis de resultados en función de la FO

Se analizarán los distintos valores de la FO obtenidos para cada conjunto de instancias atendiendo a los parámetros número de trabajos, número de máquinas y porcentaje de missing.

En la tabla 6 pueden encontrarse los promedios del factor ARPD para las soluciones obtenidas en ambos algoritmos clasificándolas según el número de trabajos. Además, se muestra el valor del factor ARPD para, la secuencia inicial del algoritmo, por lo que también se podría comprobar cuán mejor es la solución proporcionada por cada uno de los algoritmos respecto al valor de la FO en la secuencia inicial.

núm. Trabajos	4	8	16
Secuencia inicial	11,72%	11,38%	10,80%
Algoritmo 1	2,54%	1,60%	0,92%
Algoritmo 2	0,54%	1,32%	1,51%

Tabla 6: *ARPD en función del número de trabajos.*

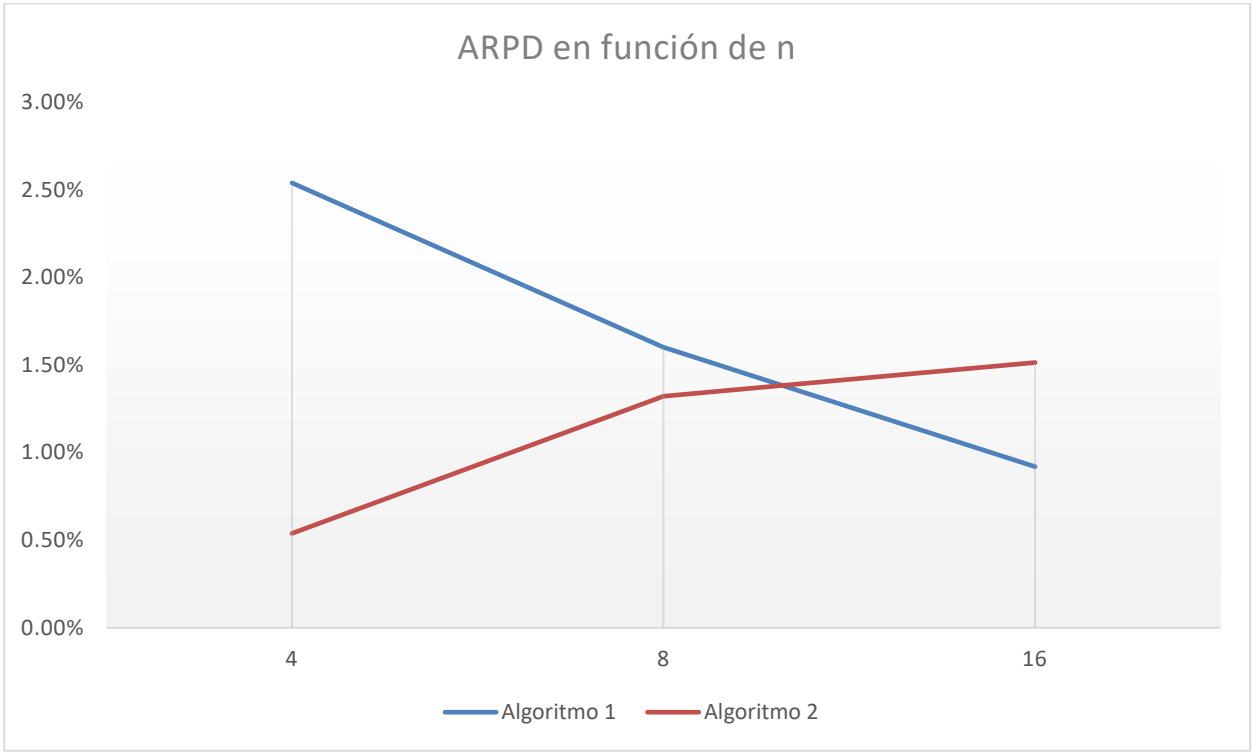

Figura 15: *ARPD en función del número de trabajos*

En la tabla 7 pueden encontrarse los promedios del factor ARPD para las soluciones obtenidas en ambos algoritmos clasificándolas según el número de máquinas. Además, se muestra el valor del factor ARPD para la secuencia inicial del algoritmo, por lo que también se podría comprobar cuan mejor es la solución proporcionada por cada uno de los algoritmos respecto al valor de la FO en la secuencia inicial.

núm. Máquinas	7→1	14→1	28→1
Secuencia inicial	13,33%	11,97%	8,63%
Algoritmo 1	2,09%	1,62%	1,35%
Algoritmo 2	1,04%	1,24%	1,09%

***Tabla 7**: ARPD en función del número de máquinas.*

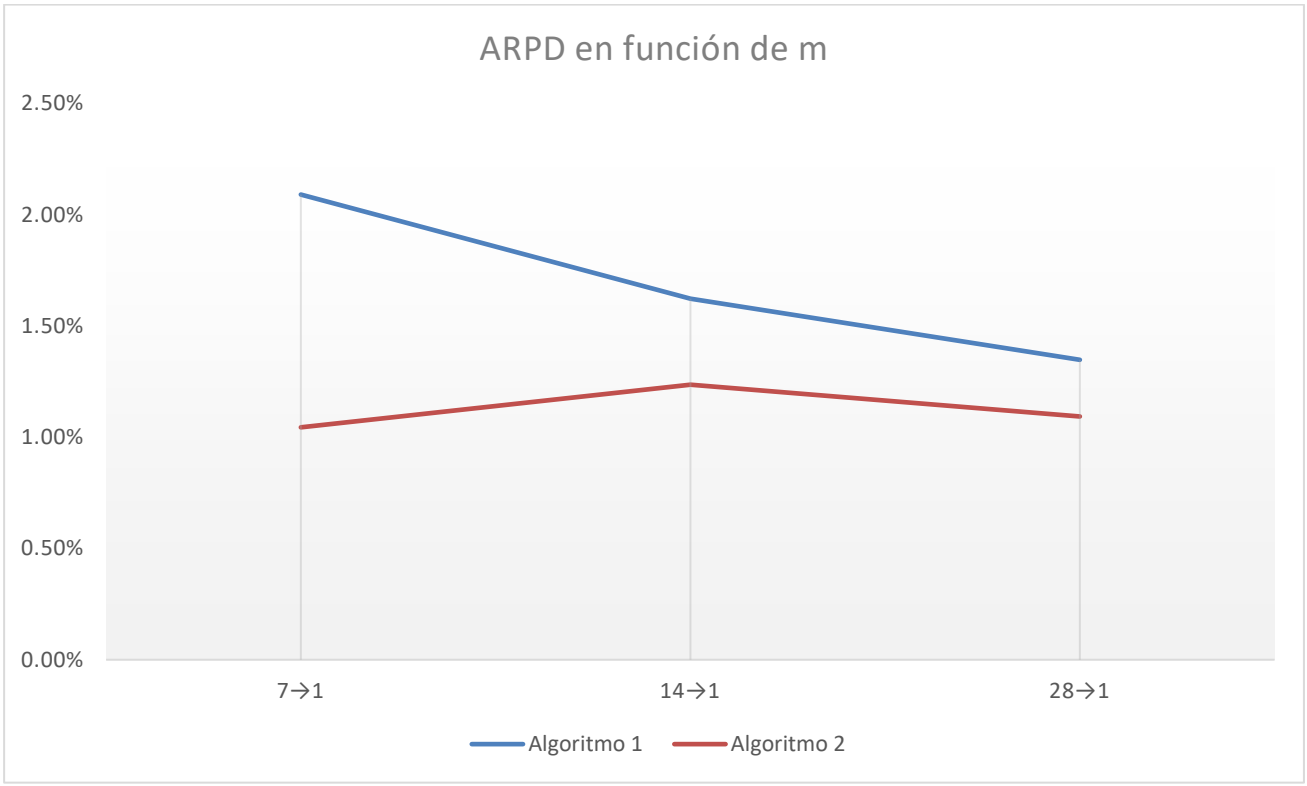

***Figura 16:** ARPD en función del número de máquinas*

En la tabla 8 pueden encontrarse los promedios del factor ARPD para las soluciones obtenidas en ambos algoritmos clasificándolas según el % de missing. Además, se muestra el valor del factor ARPD para la secuencia inicial del algoritmo, por lo que también se podría comprobar cuan mejor es la solución proporcionada por cada uno de los algoritmos respecto al valor de la FO en la secuencia inicial.

% missing	0%	25%	50%	60%
Secuencia inicial	7,99%	11,55%	13,78%	11,31%
Algoritmo 1	1,34%	1,67%	1,87%	1,80%
Algoritmo 2	0,87%	1,00%	1,38%	2,74%

Tabla 8: *ARPD en función del número del porcentaje de missing.*

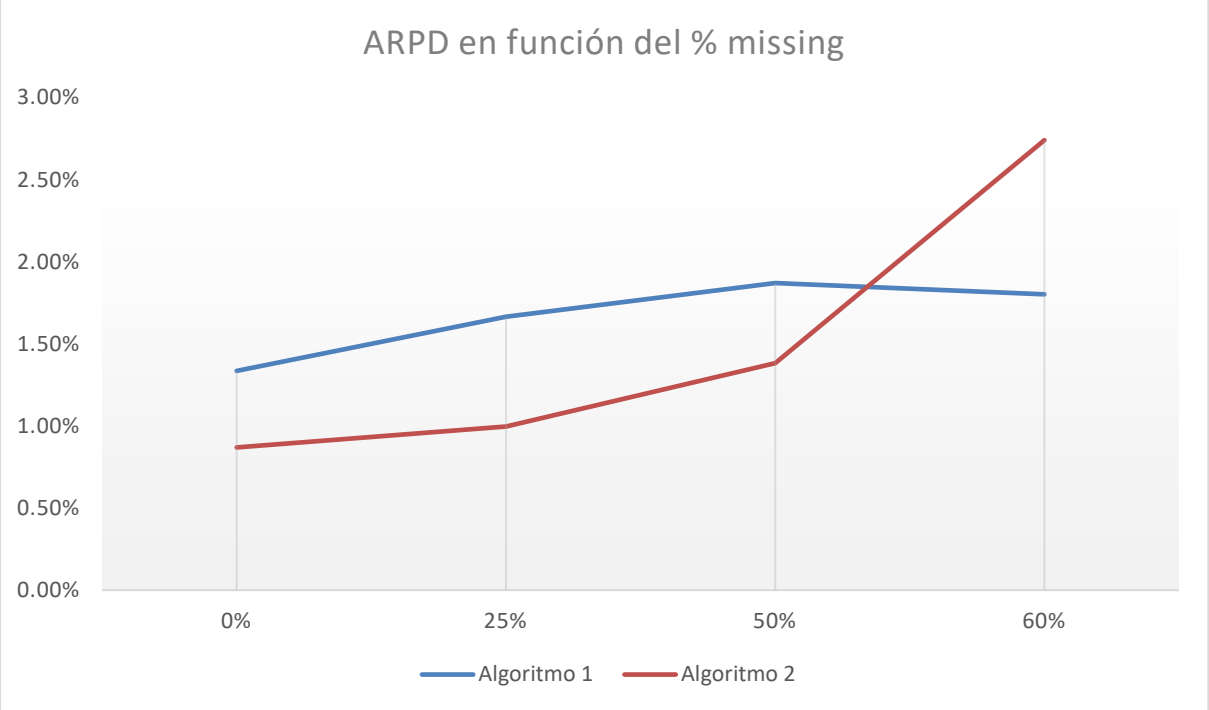

Figura 17: *ARPD en función del número del porcentaje de missing.*

Por último, en la tabla 9, se puede observar de media cómo de óptimas son las soluciones de un algoritmo respecto al otro.

	Secuencia inicial	Algoritmo 1	Algoritmo 2
ARPD	11,32%	1,69%	1,12%

Tabla 9: *ARPD promedio de ambos algoritmos*

Con la ayuda de todos los datos tablas y gráficas sobre el análisis de resultados de la FO según distintos parámetros se pueden elaborar conclusiones sobre la eficacia de ambos algoritmos.

A simple vista, gracias a la tabla 9, se puede ver que, por norma general, el algoritmo 2 arroja mejores soluciones que el algoritmo 1. Sin embargo, hay que tener en cuenta que la diferencia de valores entre ambos no es lo demasiado grande como para afirmar rotundamente que algoritmo es mejor.

Según lo indicado en la gráfica 15, se puede ver que el algoritmo 1, arroja mejores soluciones cuanto mayor es el número de trabajos a procesar. Para el algoritmo 2, se puede observar una caída en la calidad de las soluciones que aporta a medida que se va aumentando el número de trabajos.

Por lo tanto, de esta gráfica se puede extraer la conclusión de que el algoritmo 1 da buenas soluciones a medida que aumentan el número de trabajos a procesar, mientras que el algoritmo 2 aporta soluciones bastante buenas comparadas con el algoritmo 1 cuando se trata de procesar un pequeño número de trabajos.

Respecto al desempeño de los algoritmos respecto al número de máquinas, se puede comprobar en la figura 16 que el valor de este parámetro no tiene una influencia significativa en los valores de las soluciones obtenidas por ambos algoritmos. Debido a la estacionalidad de los datos, estos no indican ninguna tendencia, por lo que se podría extraer la conclusión de que el número de máquinas no afecta en la calidad de las soluciones aportadas por ambos algoritmos.

Finalmente, comparando ambos algoritmos en función del % de missing, podemos observar como la calidad de las soluciones aportada por el algoritmo 1 no se ve afectada por los valores de este parámetro. Sin embargo, el algoritmo 2 obtiene mejores soluciones para % de missing nulos o bastante pequeños, mientras que a medida que aumenta este porcentaje obtiene soluciones cada vez peores.

Por lo tanto, se podría decir que el algoritmo 1 proporciona buenas soluciones cuando se está trabajando con un número grande de trabajos a procesar, a pesar de necesitar un número mayor de iteraciones. El algoritmo 2, proporciona mejores soluciones por lo general que el algoritmo 1, exceptuando los casos con porcentajes de missing elevados.

6.2. Análisis de resultados en el problema estudiado

El problema a resolver, cuenta con los siguientes parámetros:

- Número de máquinas: 7→1
- Número de trabajos: 4
- % de missing: 53,57%

Haciendo referencia al número de máquinas, cualquiera de los dos algoritmos podría resultar óptimo. Según el número de trabajos, el algoritmo 2 arroja mejores soluciones cuando se trabaja con un número de trabajos pequeño. Sin embargo, a mayor % de missing el algoritmo 1 comienza a proporcionar mejores soluciones.

Si nos fijamos en la gráfica de la figura 17, podemos comprobar, que para un % de missing del 53%, la calidad de las soluciones del algoritmo 1 serán ligeramente superiores, aunque ambas soluciones arrojaran soluciones bastante parecidas.

Tras la ejecución del programa descrito en el capítulo 4, se obtienen las siguientes soluciones del problema:

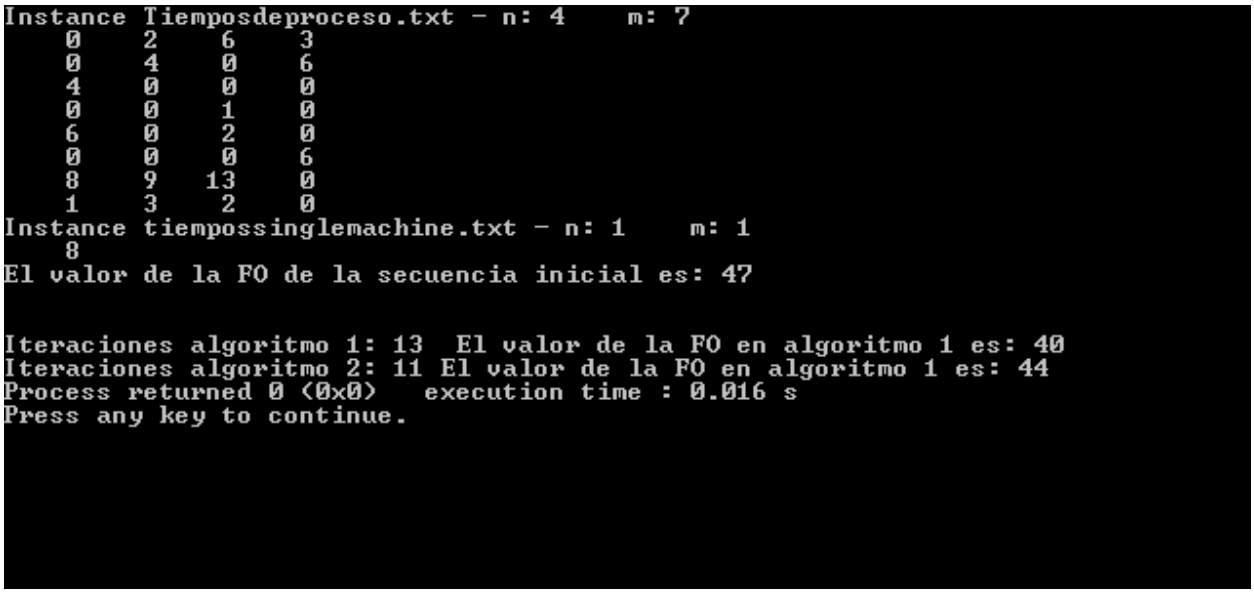

Figura 18: *Resultados computacionales del problema estudiado*

	FO	n. iteraciones
Algoritmo 1	40	13
Algoritmo 2	44	11

Tabla 10: *Soluciones aportadas por ambos algoritmos para el problema inicial*

En la figura 18, se puede observar el valor de FO objetivo para la secuencia inicial es de 47. Si se analizan las soluciones obtenidas para cada uno de los algoritmos, se puede comprobar que el algoritmo 1 mejora el C_{max} de la secuencia inicial en un 17,50%. Mientras que el algoritmo 2 la mejora en un 6,82%.

Por tanto, coincidiendo con las conclusiones obtenidas en apartados anteriores, se podría afirmar que el algoritmo 2 proporciona en general mejores soluciones que el algoritmo 1, exceptuando aquellos problemas en los cuáles el % de missing sea superior al 50%.

El algoritmo 1, encuentra una solución mejor para este problema, a pesar de necesitar un mayor número de iteraciones hasta llegar a ella.

7. CONCLUSIONES

"Saltar rápidamente a conclusiones rara vez conduce a felices aterrizajes".
Mitchell Siporin

EN la actualidad, la globalización de los mercados los hace cada vez más competitivos. El comprador posee cada vez más opciones de compra para un mismo producto, lo que hace, que sea necesario ofrecer productos de mayor calidad, confiabilidad y con un menos precio a los ofrecidos por la competencia.

Por tanto, la optimización de procesos está tomando una importancia mucho mayor en la actualidad de la que pudo tener en décadas anteriores. Uno de los procesos en los cuáles la optimización cobra una enorme importancia dentro de una empresa, es en la fabricación de sus productos. Fabricar de manera eficaz y eficiente permite obtener un mayor volumen productivo a la vez que disminuyen los costes de fabricación del producto.

La programación y optimización de la producción se lleva a cabo mediante decisiones a corto plazo, por lo que se recurre técnicas de secuenciación de la producción que proporcionen buenas soluciones en un tiempo razonable. La secuenciación de la producción podría definirse, por tanto, como la una respuesta operativa para optimizar la producción de un bien o servicio.

Uno de los entornos que está tomando mayor importancia hoy en día, son los entornos tipo assembly. Estos entornos se dan en industrias y servicios en los cuales una actividad solo puede comenzar cuando una serie de tareas previas han sido completadas. Uno de los ejemplos de porque estos entornos son cada vez más usados en la industria, son las empresas que optan por la subcontratación de la fabricación de algunas de las distintas partes del producto final para la reducción de costes. Estas piezas, deben ser ensambladas posteriormente.

Este trabajo, se ha centrado en un caso real de una empresa dedicada a la producción de moldes para la posterior fabricación de moldes plásticos. El entorno que se ha identificado en la empresa es un entorno en dos fases.
La primera de ella se corresponde con un flowshop, existen una serie de trabajos que deben ser procesados por las máquinas en un determinado orden, común para todos ellos. Cada trabajo solo puede ser procesado por una máquina al mismo tiempo, y cada máquina solo puede procesar un

trabajo en el mismo instante. Según el estudio realizado en la empresa real, se han supuesto los desplazamientos de los trabajos entre las distintas máquinas iguales a 0. También, se ha supuesto un almacenamiento a pie de máquina infinito. Sin embargo, este primer entorno cuenta con la peculiaridad de que no todos los trabajos han de ser procesados por todas las máquinas, esta restricción se identifica como *% missing.* El % de missing, indica el porcentaje de trabajos que no pasan por alguna de las máquinas.

La segunda etapa, corresponde a la fase de assembly. Una vez el conjunto de trabajos han sido procesados en la primera etapa, y solo cuando todos ellos han sido procesados íntegramente, se procesan en la segunda fase. En esta segunda fase, se observa un entorno *single machine* en el cual la máquina tiene el objetivo de ensamblar todo el conjunto de piezas.

Para la solución de este problema, se ha recurrido a la implementación de dos algoritmos distintos que puedan dar una secuencia lo suficientemente óptima para él. Ambos algoritmos se basaban en la metaheurística *Iterated Local Search (ILS)* o búsqueda local iterada. La principal diferencia entre ellos, es que el primer algoritmo realiza un estudio de la vecindad mediante inserción, mientras que el segundo lo realiza por intercambio adyacente. Para ambos, se introduce una perturbación mediante intercambio para evitar los óptimos locales y una mejora mediante *Best Improvement.*

Antes de implementarlos en el problema estudiado, se ha generado una batería de 360 problemas para estudiar la eficacia y eficiencia de ambos. Cada una de las 360 instancias, proporcionaba valores distintos de los parámetros número de trabajos, número de máquinas y % de missing.

Del estudio posterior de los resultados computacionales de estos, se puede extraer la conclusión de que, por norma general, el algoritmo 2 proporciona soluciones de mayor calidad y en un menor número de iteraciones que el algoritmo 1. Sin embargo, a medida que aumenta el % de missing, el algoritmo 1 proporciona soluciones con una calidad superior a las del algoritmo 2, a pesar de necesitar un mayor número de iteraciones para ello.

En el problema estudiado, el % de missing es bastante elevado, por lo que el algoritmo 1 es el que proporciona mejores soluciones para el problema. Esto se puede comprobar fácilmente al comprobar que el algoritmo uno mejora el valor del *makespan* en un 17,5%, mientras que el algoritmo 2 tan solo lo hace en un 6,82%.

8. BIBLIOGRAFÍA

Michael L. Pinedo: “Scheduling Theory, Algorithms, and Systems” [2011]

Xingye Dong; Houkuan Huang, Ping Chen: “An iterated local search algorithm for the permutation flowshop problem with total flowtime criterion”, School of Computer and Information Technology, Beijing Jiaotong University, 100044 Beijing, China [2008]

Katsundo hitomi: “Manufacturing Systems Engineering: A unified approach to manufacturing technology, production management, and industrial economics”, Ryukoku University and Nanjing University [1996]

Angel A. Juana; Helena R. Lourencob; Manuel Mateoc; Quim Castella; Barry B. Barriosa: “ILS-ESP: an efficient, simple, and parameter-free algorithm for solving the permutation flow-shop problem”, Dep. de Organización de Empresas, Universidad Politécnica de Cataluña [2012]

Jose M. Framiñan; Paz Perez-Gonzalez; Victor Fernandez-Viagas: “Deterministic Assembly Scheduling Problems: A Review and Classification of Concurrent-Type Scheduling Models and Solution Procedures” [2018]

David de la Fuente Nazario García; Alberto Gómez Javier Puente: “Organización de la Producción en Ingenierías”, Universidad de Oviedo [2006]

Gastón Enrique Palleiro: “Planificación, Programación y Control de la Producción” [2012]

Ali Allahverdi; Fawaz S. Al-Anzi: “The two-stage assembly flowshop scheduling problema with bitcreteria of makespan and mean completion time”, Kuwait University [2007]

Pedro J. Copado Méndez: “Hibridación de la Metaheurística ACO para el Shortest Capacitated Paths Problem”, Universidad Politécnica de Cataluña [2008]

Framiñan, Jose M; Leisten, R; Ruiz García, R.: “Manufacturing Scheduling System. An Integrated View on Models, Methods and Tools” [2014]

Steven Nahmias: “Análisis y Producción de las Operaciones” [2014]

Jose M Framiñan; Rainer Leisten; Rubén Ruiz: “Manufacturing Scheduling Systems” [2013]

Lluis Cuatrecasas Arbós: “Organización de la Producción y Dirección de Operaciones: sistemas actuales de gestión eficiente y competitiva” [2011]

José Francisco Chicano García: “Tesis doctoral: Metaheurísticas e Ingeniería del Software”, Universidad de Málaga [2007]

8. ANEXOS

8.1. Código en C

```
#include <stdio.h>
#include <stdlib.h>
#include <C:\Users\ANGELA\Desktop\TFG\libreria\schedule_lib.h>

struct instancia
{
    int n;
    int m;
    int ps;
    float P;
    long int FO;
    MAT_INT Pij;
    VECTOR_INT seq;
};

int FO_Cmax (MAT_INT p_ij, int n1, int m1, VECTOR_INT seq, int p_single);
void generarInstancia();
int IteratedLocalSearch1 (MAT_INT p_ij, VECTOR_INT seq, int n, int m, int
p_single);
int IteratedLocalSearch2 (MAT_INT p_ij, VECTOR_INT seq, int n, int m, int
p_single);

int main()
{
    int n1;
    int m1;
    int n2;
    int m2;
    int FO_inicial;

    ///PARA CALCULAR OPTIMO PARA EL PROBLEMA REAL
    MAT_INT p_ij_flow; //Cargamos tiempos de proceso de fase flowshop
    p_ij_flow = loadPTimes_nrows("Tiemposdeproceso.txt",&n1, &m1, YES);

    VECTOR_INT seq = DIM_VECTOR_INT(n1);
    randSequence(seq,n1);

    MAT_INT p_ij_single; //Cargamos tiempos de proceso de fase single
    machine
    p_ij_single = loadPTimes_nrows("tiempossinglemachine.txt", &n2, &m2,
    YES);

    int p_single;
    p_single = p_ij_single[0][0]; //Como es single machine almacenamos el
    tiempo de proceso en una variable

    FO_inicial = FO_Cmax(p_ij_flow, n1, m1, seq, p_single);
```

```
      printf("El valor de la FO de la secuencia inicial es: %d \n",
      FO_inicial);

       IteratedLocalSearch1(p_ij_flow, seq, n1, m1, p_single);
      IteratedLocalSearch2(p_ij_flow, seq, n1, m1, p_single);

    ///FIN CALCULAR PARA EL PROBLEMA REAL

    ///CALCULAMOS FO Y NUM DE ITERACIONES PARA LAS 360 INSTANCIAS ALEATORIAS

    struct instancia *instancias_main = malloc(360*sizeof(struct
    instancia));
    generarInstancia(instancias_main);
    int i;
    VECTOR_INT res_algoritmo0 = DIM_VECTOR_INT(360);
    VECTOR_INT res_algoritmo1 = DIM_VECTOR_INT(360);
    VECTOR_INT res_algoritmo2 = DIM_VECTOR_INT(360);
    for(i=0; i<360; i++)
    {
          res_algoritmo0[i] = FO_Cmax(instancias_main[i].Pij,
          instancias_main[i].n, instancias_main[i].m,
          instancias_main[i].seq, instancias_main[i].ps);
          res_algoritmo1[i] = IteratedLocalSearch1(instancias_main[i].Pij,
          instancias_main[i].seq, instancias_main[i].n,
          instancias_main[i].m, instancias_main[i].ps);
          res_algoritmo2[i] = IteratedLocalSearch2(instancias_main[i].Pij,
          instancias_main[i].seq, instancias_main[i].n,
          instancias_main[i].m, instancias_main[i].ps);
    }

    ///GUARDAMOS LOS RESULTADOS EN UN FICHERO

    FILE *resultados;
    resultados=fopen("Resultados.txt", "a");
    for(i=0; i<360; i++)
    {
          fprintf(resultados, "\nInstancia %d resultados: Algoritmo 1:: %d
          ::Algoritmo2:: %d", i, res_algoritmo1[i], res_algoritmo2[i]);
    }
    fclose(resultados);
    free(instancias_main);
    return 0;
}


int FO_Cmax(MAT_INT p_ij_flow, int n1, int m1, VECTOR_INT seq, int p_single)
{
    long int C_max_flowshop;
    long int C_max;

    C_max_flowshop = Fm_Prmu_Cmax(seq, n1, m1, p_ij_flow);
    C_max = C_max_flowshop + p_single;

    return C_max;
}
```

```
int IteratedLocalSearch1 (MAT_INT p_ij, VECTOR_INT seqOr, int n, int m, int
p_single) //Metodo aproximado 1 para resolverlo. Estudiando por inserción,
best improvement.
  {
      int FO_opt;
      int FO_iter;
      int FO_vecino;
      int i;
      int j;
      int aux = 0;
      int iter = 0;
      int iter_opt =0;
      VECTOR_INT seq = DIM_VECTOR_INT(n);
      copyIVector(seqOr,seq,n);
      VECTOR_INT seq_opt = DIM_VECTOR_INT(n);
      VECTOR_INT vecinos = DIM_VECTOR_INT(n);
      VECTOR_INT seq_iter= DIM_VECTOR_INT(n);
      VECTOR_INT seq_iter_opt = DIM_VECTOR_INT(n);
      VECTOR_INT seq_perturbacion = DIM_VECTOR_INT(n);

      FO_iter = FO_Cmax(p_ij, n, m, seq, p_single); //Inicializamos valores de
      la iteración y del problemacon los de la secuencia que se recibe
      FO_opt = FO_iter;


      copyIVector(seq, seq_opt, n); //Inicializamos todas las secuencias a la
      única que se tiene
      copyIVector(seq, seq_iter, n);
      copyIVector(seq, seq_iter_opt, n);
      printf("\n");
      while(aux<10)
      {
          iter++;
          for(i=0; i<n; i++)
          {
              int a = seq[i];     //a será el valor que se va a ir insertando.
              extractIVector(seq_iter, n, i);
              copyIVector(seq_iter, vecinos, n-1);

              for(j=0; j<n; j++)
              {
                  insertIVector(vecinos, n, a, j);
                  //print_int_vector(vecinos,n);
                  FO_vecino = FO_Cmax(p_ij, n, m, vecinos, p_single);

                  if(FO_vecino<=FO_iter)
                  {
                      FO_iter=FO_vecino;
                      copyIVector(vecinos, seq_iter_opt, n);
                  }
                  copyIVector(seq_iter, vecinos, n-1);
                  extractIVector(vecinos,n,j);
              }
              copyIVector(seq,seq_iter, n);
          }
```

```
        if(FO_opt>FO_iter) //Comparamos la FO antes de estudiar la vecindad
        con la obtenida tras la iteracion.
        {
            FO_opt=FO_iter;
            copyIVector(seq_iter_opt, seq_opt, n);
            copyIVector(seq_opt, seq, n);
            iter_opt = iter;
        }
        else // PERTURBACION CUANDO NO SE ENCUENTRA NINGUNA SOLUCION OPTIMA
        {
            int posAleatoria2,aux2;
            int posAleatoria1;
            posAleatoria1 = rand()%n;
            do
            {
                posAleatoria2 = rand()%n;
            }
            while(posAleatoria1==posAleatoria2);

            aux2 = seq_iter[posAleatoria1];
            seq_iter[posAleatoria1] = seq_iter[posAleatoria2];
            seq_iter[posAleatoria2] = aux2;
            aux=aux+1;
        }
    }

    FILE *resultadosalgoritmo1;
    resultadosalgoritmo1=fopen("Resultadosalgoritmo1.txt", "a");

    fprintf(resultadosalgoritmo1, "\nIteraciones algoritmo 1: %d ", iter);

    fclose(resultadosalgoritmo1);

    return FO_opt;
 }


 int IteratedLocalSearch2 (MAT_INT p_ij, VECTOR_INT seqOrigen, int n, int m,
int p_single)
 {
    int FO_opt;
    int i;
    int j;
    int aux;
    int bucle=0;
    int iter = 0;
    int iter_opt = 0;

    VECTOR_INT seq_opt = DIM_VECTOR_INT(n);
    VECTOR_INT seq = DIM_VECTOR_INT(n);
    copyIVector(seqOrigen,seq,n);
    VECTOR_INT seqRandom = DIM_VECTOR_INT(n);
    randSequence(seqRandom,n);
```

```
      FO_opt = FO_Cmax(p_ij, n, m, seqRandom, p_single); //Inicializamos
valores de la iteración y del problema con los de la secuencia que se recibe

      copyIVector(seq, seq_opt, n); //Inicializamos todas las secuencias a la
      única que se tiene
      int FOseq;
      int FOseqM;
      int mejorado=0;
      while(bucle<10)
      {
          iter++;
          //printf("\nInicial\n");
          //print_int_vector(seq,n);
          for(i=0; i<n-1; i++)
          {
              aux = seq[i];
              seq[i]= seq[i+1];
              seq[i+1]=aux;

              FOseq = FO_Cmax(p_ij, n, m, seq, p_single);

              if(FOseq<FO_opt)
              {
                  mejorado = 1;
                  FO_opt = FOseq;
                  copyIVector(seq, seq_opt, n);
              }
              aux = seq[i];
              seq[i]= seq[i+1];
              seq[i+1]=aux;
          }

          if(mejorado==1)
          {
              copyIVector(seq_opt,seq,n
              iter_opt = iter;
              mejorado=0;
          }
          else
          {
              int posAleatoria2,aux2;
              int posAleatoria1 = rand()%n;
              do
              {
                  posAleatoria2 = rand()%n;
              }
              while(posAleatoria1=posAleatoria2);
              aux2 = seq[posAleatoria1];
              seq[posAleatoria2] = seq[posAleatoria1];
              seq[posAleatoria1] = aux2;

              bucle++;

          }
       }
```

```
    FILE *resultadosalgoritmo2;
    resultadosalgoritmo2=fopen("Resultadosalgoritmo2.txt", "a");
    fprintf(resultadosalgoritmo2, "\nIteraciones algoritmo 2: %d ", iter);
    fclose(resultadosalgoritmo2);

    return FO_opt;
}


void generarInstancia(struct instancia instancias[360])
{
    int i,j,k,l,trabajos,maquinas,posX,posY;
    int M[3] = {7,14,28};
    int N[3] = {4,8,16};
    float MS[4] = {0,0.25,0.50,0.60};
    float ponerCero;

    int in=0;

    for(i=0; i<3; i++)
    {
        for(j=0; j<3; j++)
        {
            for(k=0; k<4; k++)
            {
                for(l=0; l<10; l++)
                {
                      instancias[in].m=M[i];
                      instancias[in].n=N[j];
                      instancias[in].P=MS[k];
                      instancias[in].ps=rand()%99+1;
                      instancias[in].seq = DIM_VECTOR_INT(instancias[in].n);
                       instancias[in].Pij =
                       DIM_MAT_INT(instancias[in].m,instancias[in].n);
                      randSequence(instancias[in].seq, instancias[in].n);
                      printf("\nSecuencia de la instancia %d\n", in);
                       print_int_vector(instancias[in].seq,
                       instancias[in].n);
                      for(trabajos=0; trabajos<instancias[in].n; trabajos++)
                      {
                            for(maquinas=0; maquinas<instancias[in].m;
                            maquinas++)
                            {
                            instancias[in].Pij[maquinas][trabajos] =
                            rand()%99+1;
                        }
                    }
                      ponerCero = (int)( instancias[in].P *
                      (instancias[in].m*instancias[in].n));
                    do
                    {
                        posX=rand()%instancias[in].m;
                        posY=rand()%instancias[in].n;
                        if(instancias[in].Pij[posX][posY]>0 && ponerCero>0)
```

```
                        {
                            instancias[in].Pij[posX][posY]=0;
                            ponerCero--;
                        }
                    }
                    while(ponerCero>0);
                    in++;
                }
            }
        }
    }

}
```