Trabajo Fin de Grado
Grado en Ingeniería de Organización Industrial

Metaheurísticas aplicadas al problema de Flowshop de permutación con dos conjuntos de trabajo

Autor: Isaac Fernández Montilla
Tutora: Paz Pérez González

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

Sevilla, 2020

Trabajo Fin de Grado
Grado en Ingeniería de Organización Industrial

Metaheurísticas aplicadas al problema de Flowshop de permutación con dos conjuntos de trabajo

Autor:

Isaac Fernández Montilla

Tutor:

Paz Pérez González

Profesor titular

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

Sevilla, 2020

Trabajo Fin de Grado: Metaheurísticas aplicadas al problema de Flowshop de permutación con dos conjuntos de trabajo

Autor: Isaac Fernández Montilla

Tutora: Paz Pérez González

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

A mi familia

A mis maestros

A mis amigos

Agradecimientos

Me gustaría dar las gracias a varias personas por su ayuda durante la realización de este trabajo de fin de grado.

A mi tutora Paz Pérez González, por aceptar mi petición para tutorarme en este proyecto, así como por su tiempo y dedicación durante todo el proceso.

A mi familia y amigos por ayudarme y darme ánimos siempre durante todos estos años y por estar siempre a mi lado cuando los necesitaba.

A mi apoyo incondicional, Lara, porque sin su ayuda no hubiese terminado todo esto y sobretodo, nada sería igual sin ella.

A todos ellos, gracias.

Isaac Fernández Montilla

Sevilla, 2020

Resumen

En este Trabajo de Fin de Grado, se aboradará el problema de taller de flujo regular para dos conjuntos de trabajo. En problemas de programación con dos agentes, se deben programar los dos conjuntos de trabajos, cada uno con una función objetivo a minimizar. Se pueden considerar diferentes enfoques multicriterios. En este caso, se aplica el enfoque *épsilon-constraint*, minimizando el objetivo del primer agente, mientras que el objetivo del segundo agente debe ser menor que un límite superior dado. El objetivo final será el de obtener la mejor solución posible para este problema de programación de la producción considerando como entorno un flowshop de permutación.

Para resolver el problema aplicaremos una metaheurística, que se denominará Variable Neighbourhood Search o Búsqueda en vecindad variable (VNS). Haciendo uso de dicha metaheurística se tratará de encontrar la mejor solución posible para el problema planteado mediante el estudio de tres tipos de vecindades. En este caso, el objetivo será el mismo para cada conjunto de trabajos, considerandose la minimización del tiempo total de terminación ponderado. Con este objetivo, si cada conjunto de trabajos pertenece a diferentes clientes y cada trabajo tiene una prioridad distinta, pretende minimizar el tiempo medio de terminación de los trabajos de cada conjunto teniendo en cuenta dichas prioridades.

Para llevar a cabo toda programación se usará el lenguaje C y luego se analizarán los resultados con la herramienta Microsoft Excel.

Abstract

In this Final Degree work, we will consider the two-agent permutation flowshop scheduling problem. In two-agents scheduling problems, two set of jobs should be scheduled, each one with an objective function to be minimized. Different multicriteria approaches can be considered. In this case, the epsilon-constraint approach is applied, minimizing the objective of the first agent, while the objective of the second agent should not be greater than a given upper bound. The final goal of this project will be to obtain the best possible solution for this scheduling problem.

In order to solve the problem, we will apply a metaheuristic, called Variable Neighbourhood Search (VNS). Making use of this metaheuristic, we will try to find the best possible solution for our problem by studying three types of neighbourhoods. In this case, the objective function is to minimize the total weighted completion time of the two agents.

To carry out all this programming, C language will be used, and then we will analyze all the results using Microsoft Excel.

Índice

Agradecimientos **ix**

Resumen **xi**

Abstract **xiii**

Índice de Tablas **xvi**

Índice de Figuras **xviii**

1 Introducción **1**
1.1 Conceptos básicos *2*
1.2 Notación *3*
1.3 Clasificación de los modelos de programación de la producción *4*
1.3.1 Características de las máquinas. Tipos de entornos (α) 4
1.3.2 Características de los trabajos. Restricciones (β) 5
1.3.3 Función objetivo (γ) 6
1.4 Métodos de programación de la producción *7*
1.5 Descripción del problema *7*

2 Metaheurísticas aplicables **11**
2.1 Tipos de metaheurísticas *11*
2.1.1 Metaheurísticas basadas en trayectoria 11
2.1.2 Metaheurísticas basadas en población 12
2.2 Variable Neighbourhood Search *13*
2.2.1 Variable Neighbourhood Descent (VND) 15
2.2.2 Reduced Variable Neighbourhood Search (RVNS) 15

3 Método de resolución **17**
3.1 Cálculo de la Función Objetivo *17*
3.2 Cálculo de las vecindades *19*
3.2.1 General Swap 19
3.2.2 Insertion 20
3.2.3 Learner 21
3.3 Variable Neighbourhood Search *22*
3.3.1 Solución inicial 22
3.3.2 VNS 23

4 Análisis computacional **25**
4.1 Generación de instancias *25*
4.2 Análisis de resultados *26*

5 Conclusiones **31**

6 Referencias **33**

7 Anexo **35**
7.1 Código en c *35*
7.2 Código de generación de instancias *52*

ÍNDICE DE TABLAS

Tabla 1: Problema generado para n=20 y m=5. 26

Tabla 2: ARDI de cada problema para cada delta. 27

Tabla 3: ARDI en función del número de tabajos. 28

Tabla 4: ARDI en función del valor delta. 29

ÍNDICE DE FIGURAS

Figura 1. Representación de un programa, Gantt Chart. 2
Figura 2. Clasificación de los modelos de programación de la producción 4
Figura 3: Esquema de máquinas paralelas. 5
Figura 4: Esquema de un taller de flujo regular (flowshop). 5
Figura 5: Clasificación de las metaheurísticas. 11
Figura 6: Intercambio (Swap). 14
Figura 7: Inserción (Insertion). 14
Figura 8: Vecindad Learner. 14
Figura 9: Pseudocódigo de la función de cálculo de la función objetivo de los trabajos del conjunto A. 18
Figura 10: Pseudocódigo de la función de cálculo del tiempo total de terminación. 18
Figura 11: Pseudocódigo de la función de cálculo de la vecindad General Swap. 20
Figura 12: Pseudocódigo de la función de cálculo de la vecindad Insertion. 20
Figura 13: Pseudocódigo de la función de cálculo de la vecindad Learner. 22
Figura 14: Pseudocódigo del cálculo de la solución inicial. 23
Figura 15: Pseudocódigo del cálculo de la metaheurística VNS. 24
Figura 16: ARDI de cada tamaño para cada delta. 27
Figura 17: ARDI de cada delta para cada tamaño. 28
Figura 18: ARDI en función del número de trabajos. 29
Figura 19: ARDI en función del valor delta. 30

1 Introducción

La mente que se abre a una nueva idea jamás volverá a su tamaño original.

- Albert Einstein-

La Programación de la Producción se define como el conjunto de procesos y técnicas cuyo objetivo es la asignación de los recursos de la empresa para la fabricación de un conjunto de productos. Estos recursos se asignan a unas tareas, cada una de las cuáles debe ser procesada de la forma correcta y en el lugar y momento adecuado de manera que se maximice la eficacia respecto a uno o más objetivos que se desean optimizar y la eficiencia de los recursos. Por otro lado, el Control de la Producción es el conjunto de mecanismos y herramientas utilizados para poder monitorizar un seguimiento del programa de producción y, cuando sea necesario, ejecutar acciones correctoras (Pérez González, Fernández-Viagas & Framiñán, 2020).

Este proyecto se encuentra en el ámbito de la Programación y Control de la Producción, que es un proceso que ha tomado mucha relevancia actualmente en el sector industrial, y que está contenido dentro de la Organización de la Producción (Production Management). La Organización de la Producción, consiste en la toma de un número elevado de decisiones a lo largo del tiempo para tratar de asegurar la máxima productividad con el mínimo coste posible (Pérez González, Fernández-Viagas & Framiñán, 2020).

Dentro de la programación de la producción no todos los problemas son iguales. La función objetivo que se quiera optimizar será muy diferente en cada problema, pudiendo ser por ejemplo la minimización del tiempo total de terminación de los trabajos o del número de trabajos que se entregan con retraso. Tanto los recursos como las tareas pueden variar mucho en función de la organización a la que se haga referencia. Por tanto, se pueden por ejemplo identificar como recursos las máquinas en un taller, las pistas de aterrizaje en un aeropuerto o las unidades de procesamiento en un entorno de computación. Las tareas asociadas con estos recursos pueden ser respectivamente las operaciones en un proceso productivo, los despegues y aterrizajes en un aeropuerto o las ejecuciones de un programa de ordenador (Ballesteros Silva, Ballesteros Riveros & Bravo Bolívar, 2013).

Los elementos principales de un modelo de programación son un número M de máquinas y un número N de trabajos. Con esto se puede elaborar el programa de producción. El objetivo de un modelo de programación es generar un programa, con una asignación de operaciones que sea factible y lo más eficiente posible o incluso óptima. En la Figura 1 se puede observar un diagrama de Gantt como ejemplo de un modelo de programación de la producción.

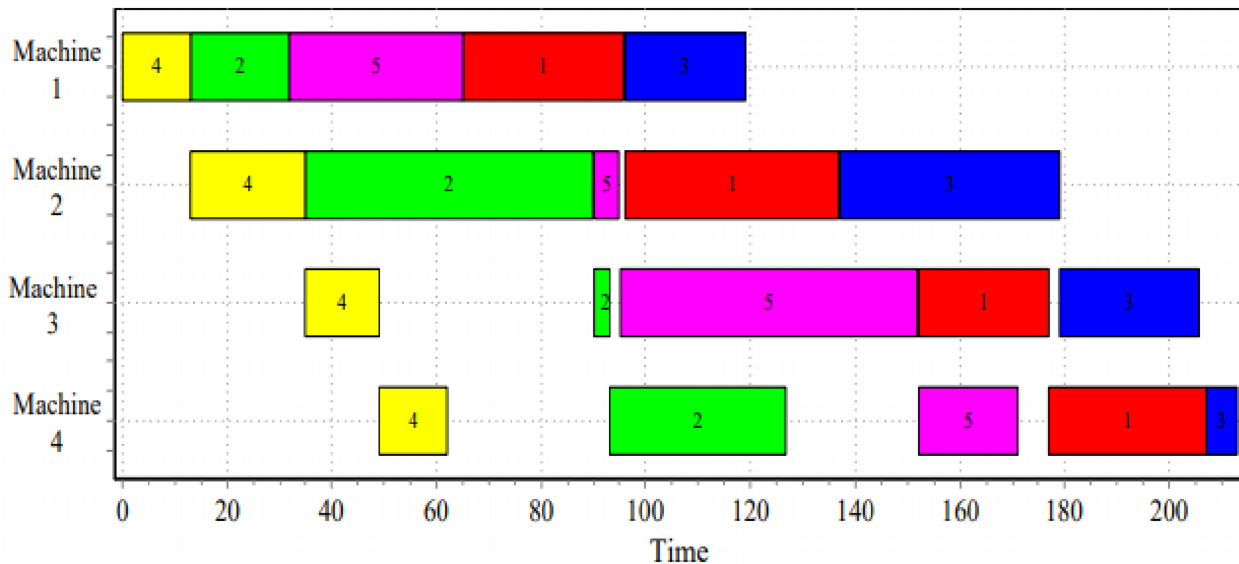

Figura 1: *Representación de un programa, Gantt Chart. Fuente: Diapositivas Programación y Control de la Producción (2020).*

1.1 Conceptos básicos

En todos los problemas de programación considerados, siempre se supone que tanto el número de trabajos como el número de máquinas son finitos. Como ya se ha expuesto anteriormente, se va a denotar con n el número de trabajos y con m el número de máquinas. También se usará habitualmente el subíndice j para hacer referencia a un trabajo y el subíndice i para una máquina. Si un trabajo pasa por una serie de procesos, el par (i, j) hace referencia a la operación del trabajo j en la máquina i. Los elementos principales, según Pérez González, Fernández-Viagas & Framiñán (2020) son los siguientes:

- **Máquina** (machine): Recurso productivo con capacidad para realizar operaciones de transformación/transporte de material.

- **Trabajo** (job): Producto que es objeto de una operación en alguna de las máquinas de la fábrica.

- **Secuencia** (sequence): Orden en el que cada trabajo comienza a procesarse en cada máquina.

- **Programa** (schedule): Asignación en la escala temporal completa de las máquinas de una empresa para la fabricación de un conjunto de trabajos. Por tanto, el programa determina el comienzo y final de cada operación a realizar en cada recurso productivo.
  Los programas pueden ser admisibles (feasible schedule) que son los que cumplen todas las características y restricciones que deben cumplir, o no admisibles que son aquellos que no cumplen alguna o todas las restricciones.
  Además de esto, se pueden distinguir tres tipos de programas admisibles:
  - **Programa semi-activo** (semi-active schedule): No es posible adelantar ninguna operación (desplazar a la izquierda en el Gantt) sin cambiar el orden en el que alguna máquina procesa los trabajos.
  - **Programa activo**: No es posible adelantar ninguna operación sin retrasar alguna otra.
  - **Programa sin retraso**: No se mantiene ninguna operación en espera mientras la máquina asignada a esta operación está disponible para procesarla.

1.2 Notación

Para la definición de todos estos elementos y su notación se ha tomado como referencia la definida por Pinedo (2012). En primer lugar, se exponen los principales datos que se van a utilizar:

- **Tiempo de proceso** (p_{ij}): Duración que requiere una operación de un trabajo j en una 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 (p_j).

- **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 se puede producir en algunas ocasiones, aunque normalmente conlleva 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 prioridad entre el conjunto de trabajos j del modelo.

Por otra parte, se tienen una serie de medidas o variables las cuáles aparecerán en los distintos problemas de programación de la producción:

- **Tiempo de terminación** o completion time (C_{ij}): representa el instante en el que el trabajo j termina de ser procesado en la máquina i. Cuando se hable del momento en el cuál el trabajo j haya sido procesado por la última máquina, se denotará como C_j.

- **Tiempo de flujo** o flowtime (F_j): representa el tiempo que el trabajo j se encuentra en el entorno. Se define como:

$$F_j = C_j - r_j$$

- **Retraso del trabajo** o lateness (L_j): representa la diferencia entre el tiempo de terminación y la fecha de entrega del trabajo j. Si el trabajo se ha completado antes de su fecha de entrega será positivo, y si no, será negativo. Se define como:

$$L_j = C_j - d_j$$

- **Tardanza del trabajo** o tardiness (T_j): representa el cuánto se ha retrasado un trabajo. En caso de que el trabajo haya sido completado antes de su fecha de entrega, su valor será 0. Se define como:

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

- **Adelanto del trabajo** o earliness (T_j): representa el cuánto se ha adelantado un trabajo. En caso de que el trabajo haya sido completado después de su fecha de entrega, su valor será 0. Se define como:

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

- **Trabajo tardío** o tardy job (U_j): expresa si un trabajo ha finalizado después de su fecha de entrega. Se define como:

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

1.3 Clasificación de los modelos de programación de la producción

Cuando se describe un modelo de programación de la producción se deben tener en cuenta varios elementos, por ello existe una clasificación en la que se definen tres conceptos. Éstos forman la siguiente notación α | β | γ (Pinedo, 2012), quedando así caracterizado cualquier problema. Cada concepto queda definido como sigue:

- α: tipo de entorno de fabricación (layout). Representa el número y la disposición de las máquinas.
- β: restricciones que caracterizan el sistema productivo.
- γ: objetivo (u objetivos) que se quiere minimizar o maximizar para obtener la solución más eficaz posible de cada modelo.

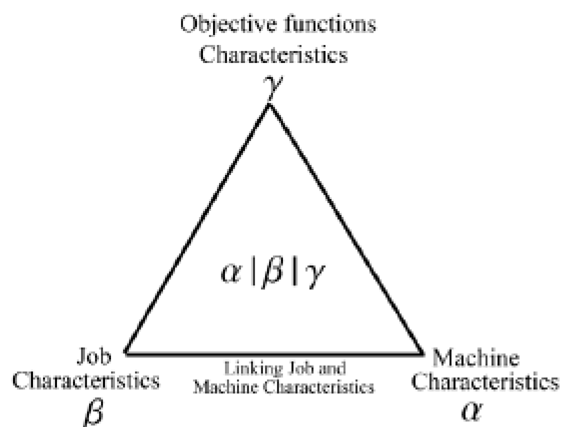

Figura 2. *Clasificación de los modelos de programación de la producción. Fuente: Diapositivas Programación y Control de la Producción (2020).*

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

La disposición y el número de máquinas es fundamental para definir y resolver problemas de programación de la producción. Existen varios tipos de entornos según Pérez González, Fernández-Viagas & Framiñán (2020):

- **Single machine** (1): todos los trabajos son procesados por una única máquina. Es el caso más simple de todos los entornos.

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

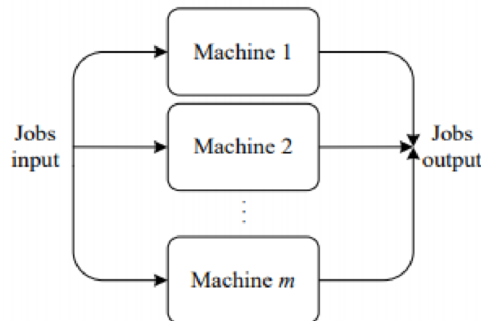

Figura 3*: Esquema de máquinas paralelas. Fuente: Diapositivas Programación y Control de la Producción (2020).*

- **Uniform parallel machines** (Qm): consiste en m máquinas en paralelo, pero cada una tiene una velocidad distinta de procesado. Esta velocidad de cada máquina se denota como v_i. El tiempo de proceso puede por tanto calcularse como:

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

- **Unrelated parallel machines** (Rm): consiste en m máquinas distintas en paralelo. El tiempo de proceso de cada trabajo depende de la máquina a la que sea asignado, ya que son diferentes en cada una.
- **Flowshop** (Fm): consiste en m máquinas en serie. Todos los trabajos deben ser procesados en todas y cada una de las m máquinas. Además, todos los trabajos tienen la misma ruta, es decir, deben pasar por las máquinas en el mismo orden.

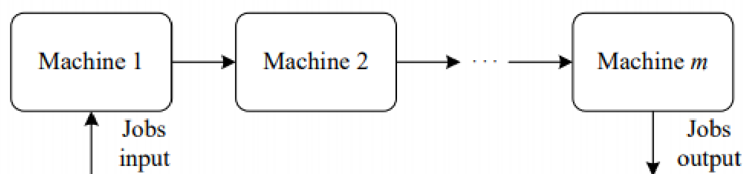

Figura 4*: Esquema de un taller de flujo regular (flowshop). Fuente: Diapositivas Programación y Control de la Producción (2020).*

- **Jobshop** (Jm): consiste en m máquinas. Cada trabajo tiene una ruta diferente y predeterminada. Esta es su diferencia con el flowshop, la ruta puede ser diferente para cada trabajo.
- **Openshop** (Om): consiste en m máquinas en las que cada trabajo debe ser procesado en todas las máquinas. Sin embargo, su peculiaridad es que no hay ruta predeterminada para cada trabajo.

1.3.2 Características de los trabajos. Restricciones (β)

Las características de los trabajos, habitualmente llamadas restricciones sirven para identificar consideraciones

del problema que no se han definido en el entorno. Las más comunes y usadas son:

- **Interrupciones** (preemption): hace referencia a si los trabajos son o no interrumpibles. Existen tres tipos de trabajos según esta restricción:
    - Pmtn−non−resumable: se pierde el trabajo hecho de la tarea interrumpida, y se empieza otra vez cuando se reinicia.
    - 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 reinicia por donde se había dejado tras la interrupción.

- **Fechas de llegada y de entrega** (r_j/d_j): el trabajo no puede comenzar a ser procesado antes de su fecha de llegada. Las fechas de entrega pueden ser restricciones cuando actúan como deadlines ($\bar{d}j$).

- **Tiempos de setup** (s_{ij}): se utilizan cuando hay que preparar la máquina para el trabajo que se va a procesar. Pueden ser anticipatorios o no anticipatorios.

- **Precedencia** (prec): esta restricción aparece cuando un trabajo no puede empezar a ser procesado hasta que no hayan terminado de procesarse los trabajos que lo preceden. Existen varios tipos de precedencia:

    - Chain precedence: cada trabajo tiene un predecesor y un sucesor.

    - Intree precedence: cada trabajo tiene un único sucesor y al menos un predecesor.

    - Outtree precedence: cada trabajo tiene un único predecesor y al menos un sucesor.

    - Tree precedence: no sigue ninguna regla.

- **Permutación** (prmu): es una restricción específica de los entornos flowshop para indicar que indicar que el orden en el que los trabajos pasan por la primera máquina se mantiene en todas las demás.

- **Máquina no ociosa** (no-idle): no están permitidos los tiempos ociosos de las máquinas entre trabajos. Una vez que la máquina empieza a procesar el primer trabajo del programa, ya no puede parar.

- **Espera no permitida** (no-wait): esta restricción indica que los trabajos no pueden esperar entre máquinas o estados.

1.3.3 Función objetivo (γ)

Todos los problemas de programación de la producción tienen como objetivo obtener la máxima eficacia posible, minimizando o maximizando unos objetivos. Las funciones objetivo más comunes tratan de minimizar los siguientes objetivos:

- **Makespan** (C_{max}): se define como el tiempo de finalización del último trabajo que sale del sistema, es decir, el tiempo total que se tarda en la producción. Un makespan bajo implica una buena utilización de la máquina.
- **Total completion time** ($\sum C_j$): se define como la suma de los tiempos de terminación de los *n* trabajos del sistema.
- **Total weighted completion time** ($\sum w_j C_j$): se define como la suma ponderada de los tiempos de finalización de los trabajos. Esta función objetivo será la que usaremos más adelante.

- **Total weighted tardiness** ($\sum w_j C_j$): se define como la suma ponderada de los retrasos.

Hay muchas más funciones objetivo que se podrían exponer, pero estas son las más usadas habitualmente, y entre ellas está la que se va a utilizar en este proyecto.

1.4 Métodos de programación de la producción

Existen muchos métodos de programación de la producción en función del modelo que se quiera resolver. Los métodos que se pueden aplicar para resolver estos problemas de programación de la producción se pueden clasificar en:

- **Métodos exactos**: son aquellos que proporcionan la solución óptima de un problema, ya que se puede asegurar que ningún otro programa va a ser más eficaz con respecto al objetivo que el obtenido mediante este método. Dentro de los métodos exactos se puede distinguir entre constructivos y enumerativos.
  Los ejemplos de constructivos más usados son:

  - Algoritmo de Johnson, para el problema F2 | | C_{max}.
  - Algoritmo de Lawler, para el problema 1 | prec | max $g(C_j)$.
  - Algoritmo de Moore, para el problema 1 | | $\sum U_j$.

  Por otro lado, se encuentran los algoritmos exactos enumerativos que son los que garantizan la evalución de todas las soluciones del modelo. Los más conocidos son:

  - Modelado MILP: consiste en la programación lineal entera mixta, y es un algoritmo mediante el cuál se puede resolver cualquier tipo de problema.
  - Branch and Bound: se utiliza para algunos problemas de optimización.

- **Métodos aproximados**: son aquellos que proporcionan un programa factible pero no pueden garantizar que sea el óptimo. Dentro de estos métodos se encuentran por un lado las heurísticas constructivas (Minimum slack, Cheapest Insertion…). Por otro están las metaheurísticas, como pueden ser Iterated Greedy, Simulated Annealing, los Algoritmos genéticos o la metaheurística que se usará en este trabajo, la VNS (Variable Neighbourhood Search).

1.5 Descripción del problema

En este apartado se va a realizar una explicación del problema concreto que se va a abordar en este trabajo. En primer lugar, se debe conocer el problema a estudiar, por lo que se debe exponer qué tipo de entorno queremos estudiar, qué restricciones presenta y qué objetivo vamos a evaluar para su resolución. Como se había dicho en el punto anterior, para describir un problema de programación de la producción, utilizamos la notación $\alpha \mid \beta \mid \gamma$ mediante la cuál quedan descritos todos los elementos del problema.

Este problema constará de un entorno de tipo taller de flujo regular, o Flowshop. En este tipo de entorno, denotado como Fm y siendo *m* el número de máquinas en cada caso, todos los trabajos pasan obligatoriamente por todas las máquinas siguiendo la misma ruta. Este caso, se va a resolver haciendo uso de dos máquinas.

En cuanto a las restricciones, solo se tiene en cuenta una muy básica:

- **pmru**: indica que es una secuencia de permutación ya que se usa la misma secuencia en todas las máquinas, es decir, los trabajos se procesan en el mismo orden en cada máquina. Esta restricción se puede aplicar sólo a los entornos Flowshop.

Por otra parte, existen una serie de suposiciones que se dan en prácticamente todos los entornos a no ser que se

indique lo contrario y también deben tenerse en cuenta, como pueden ser:

- Tiempos de transporte entre máquina se suponen despreciables.
- Cada máquina sólo es capaz de procesar un trabajo en un instante de tiempo determinado y cada trabajo sólo puede ser procesado en una máquina en cada instante de tiempo.
- Todos los trabajos están disponibles al inicio del horizonte temporal (no hay fechas de llegada).
- No es posible que se interrumpan los trabajos.
- Suponemos que el buffer entre máquinas es infinito.

Una vez especificados el entorno y la restricción, aún quedaría hablar sobre el objetivo del problema. La función objetivo que vamos a utilizar consistirá en minimizar el *Total Weighted Completion Time*, es decir, la suma ponderada de los tiempos de finalización de todos los trabajos. Una vez definidos el entorno, las restricciones y la función objetivo que se van a usar en este trabajo, debemos modelarlo finalmente para obtener el problema final que se va a resolver. Los modelos de programación con entornos flowshop son muy comunes en la práctica, más aún en la industria. Normalmente se tiene que todos los trabajos que llegan a dicho entorno provienen de un mismo agente, es decir, todos los trabajos pertenecen a un mismo conjunto de trabajos. No obstante, cuando se habla de la práctica real en un proceso industrial habitualmente hay más de un agente que proporciona tareas para un mismo fabricante. Por tanto, para adaptarlo más a la situación que se produce en la realidad, este modelo constará de dos conjuntos de trabajos diferentes.

En primer lugar, hay que decir que un problema de programación flowshop con dos conjuntos de trabajos está compuesto, como ya se había definido, de *n* trabajos que se procesan en *m* máquinas (2 máquinas en el caso que se va a estudiar). Todos estos trabajos siguen la misma ruta, es decir pasan por las máquinas en el mismo orden y son procesados primero en la máquina 1, después en la 2 y así sucesivamente. Cada máquina solo puede procesar un trabajo al mismo tiempo y estos trabajos pueden pertenecer a cualquiera de los dos conjuntos de trabajo que vamos a definir.

Explicado todo esto, el objetivo final del problema será encontrar una secuencia factible que minimice las dos funciones objetivo que se tendrán, ya que cada conjunto de trabajo tendrá su propia función objetivo. Se trabajará con el mismo objetivo para ambos, es decir, los dos conjuntos tratarán de minimizar el total weighted completion time ($\sum w_j C_j$). Como es lógico, al estar compuesto el problema por dos conjuntos de trabajos, cada conjunto afectará al otro. Por ello, no se pueden programar los trabajos de cada conjunto por separado, por lo que habrá que intentar minimizar ambas funciones objetivo al mismo tiempo. Esto hace de este problema un problema de programación multiobjetivo.

Para tratar de resolver este tipo de problema con dos conjuntos de trabajo y dos funciones objetivo que hay que minimizar al mismo tiempo, normalmente se intenta programar de tal forma que se minimice la función objetivo del primer conjunto de trabajos (Conjunto A), estableciendo un límite superior el cual la función objetivo del segundo conjunto (Conjunto B) no podrá sobrepasar. Este método, que será el que se va a usar de ahora en adelante lo es conocido como *ε-constraint method,* y es una de las metodologías más antiguas de resolución de problemas multiobjetivo, propuesta por Haimes, Lasdon y Wismer (1975). El valor de épsilon (ε) será el límite superior que la función objetivo de B no podrá superar. Esta técnica trata básicamente de convertir un modelo multiobjetivo como el que se tiene en uno mono-objetivo, escogiendo uno de los conjuntos como principal y configurando los demás como restricciones. Aunque este método tiene muchas ventajas, también tiene algún inconveniente, como por ejemplo el hecho de que sólo el objetivo del conjunto A estará siendo realmente minimizado, por lo que los resultados que se obtendrán serán más beneficiosos para este conjunto de trabajos.

En definitiva, el modelo queda de la siguiente forma:

$$\mathbf{F2 \mid prmu \mid \varepsilon\left(\sum w_j C_j^A / \sum w_j C_j^B\right)}$$

Siendo un entorno de flujo regular o flowshop de permutación de dos máquinas con dos conjuntos de trabajos A y B. Cada conjunto de trabajo tiene su propia función objetivo, aunque las dos traten de minimizar el mismo objetivo, y evidentemente no se pueden tratar independientemente. Cada conjunto de trabajos pertenece a un

cliente, y aunque en este caso los dos tienen el mismo objetivo, ambos conjuntos compiten por los recursos y por programar los trabajos de uno de los conjuntos antes. Esto, por tanto, va en contra del objetivo del otro conjunto de trabajos. Además, tenemos un valor de épsilon que servirá para llevar a cabo el método *ε-constraint* y se usará como límite superior que la función objetivo del conjunto B no podrá superar.

2 METAHEURÍSTICAS APLICABLES

Los obstáculos son esas cosas horribles que ves al apartar los ojos de la meta.

- Henry Ford-

El problema que se ha definido no ha sido muy estudiado con anterioridad, y para resolverlo se hará uso de una metaheurística de resolución. Las metaheurísticas son procedimientos generales que permiten generar soluciones aproximadas a ciertos problemas. Aunque hay un gran número de metaheurísticas, es necesario conocer y analizar muy bien el problema a resolver para determinar cuál de ellas puede ser la más adecuada. En el siguiente punto describiremos y analizaremos brevemente las metaheurísticas más usadas para la resolución de modelos.

2.1 Tipos de metaheurísticas

Cuando se habla de metaheurísticas existen diversas formas de clasificarlas, dependiendo del aspecto que se vaya a considerar. Como ejemplos, se pueden diferenciar en función de si son basadas en naturaleza o no, con o sin memoria… Sin embargo, una de las clasificaciones más populares consiste en la división de las metaheurísticas en función del número de soluciones usadas. Esta clasificación, propuesta por Chicano en su tesis doctoral (2007), divide por tanto las metaheurísticas en basadas en población y basadas en trayectoria, como podemos observar en la Figura 9.

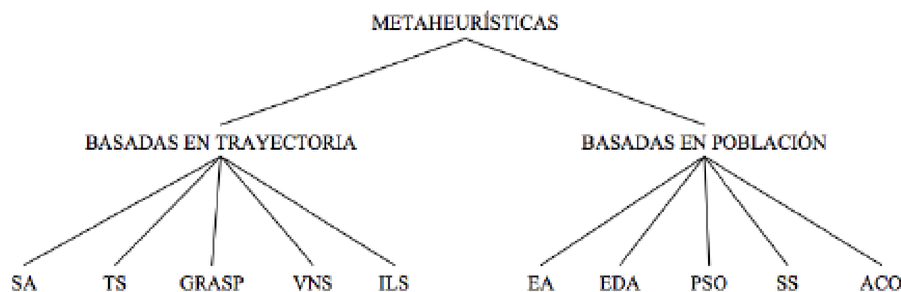

***Figura 5**: Clasificación de las metaheurísticas. Fuente: Tesis doctoral José Francisco Chicano (2007).*

2.1.1 Metaheurísticas basadas en trayectoria

Los métodos basados en trayectoria están caracterizados por partir de una solución inicial y, haciendo uso de la exploración de su vecindario, ir actualizando la solución. Estos pasos hacen que se forme una trayectoria, de ahí el su nombre. La mayoría de las metaheurísticas basadas en trayectoria vienen de las búsquedas locales

simples, ya que para escapar de un óptimo local, se aplica una perturbación a la solución obtenida en la búsqueda local. Para todos los métodos que se van a definir a continuación, es necesario definir un criterio de parada (para establecer el fin del proceso). Este criterio puede ser el establecimiento de un número de iteraciones para que cuando se llegue a la última acabe el proceso, encontrar una solución que sea muy buena o detectar un estancamiento del proceso. Las metaheurísticas basadas en trayectoria también pueden denominarse basadas en vecindad, y las más usadas (Lozano Segura, 2020) son:

- **Simulated Annealing (SA):** el método intenta simular los principios de enfriamiento de los metales. Se parte de una solución inicial a la que se le aplica una perturbación. Este método, para evitar caer en óptimos locales, acepta empeoramientos con una probabilidad que depende de T (parámetro de temperatura). El parámetro T se va modificando, por lo que al principio la probabilidad de aceptar una solución peor es alta, pero luego se va minimizando. Este método encontraría el óptimo del problema si se realizasen infinitas iteraciones, pero es un método muy lento y en caso de no realizar las infinitas iteraciones, sus resultados son poco fiables.
- **Iterated Local Search (ILS):** el método ILS parte de una solución inicial a la que se le aplica búsqueda local para llegar a un óptimo local. A continuación, se le aplica una perturbación para alejarse del óptimo local y llegar así a otro "valle", donde se vuelve a hacer búsqueda local y así sucesivamente. Existes dos criterios diferentes para elegir si un óptimo local es aceptado o no, ya que se puede seleccionar la mejor solución, o bien se puede seleccionar siempre la nueva solución, y así diversificar.
- **GRASP**: sus siglas provienen de Greedy Randomized Adaptative Search Procedure y esta metaheurística se caracteriza por ser un proceso iterativo con dos fases (constructiva y de mejora). La primera fase o fase constructiva, parte de una solución parcial vacía a la que habrá que ir añadiendo aleatoriamente elementos que se encuentran en una "candidate list". Estos elementos tienen un índice Greedy, que muestra las ventajas de añadir ese elemento. Cuando se construye la solución, se pasa a la fase de mejora, que consiste en aplicar una búsqueda local para mejorar la solución.
- **Tabu Search (TS):** es un procedimiento de búsqueda por entornos cuya característica es que hace uso de una memoria adaptativa. Utiliza una memoria a corto plazo, creando una Lista Tabú en la cuál se encuentran las últimas soluciones que se han encontrado para evitar seleccionarlas otra vez en las siguientes iteraciones. Esta técnica permite el empeoramiento, lo que se utiliza para evitar caer en óptimos locales. Cuando se obtiene una solución mejor, ésta se añade a la lista tabú.
- **Búsqueda en vecindad variable (Variable Neighbourhood Search VNS):** esta es una metaheurística que se basa en el uso de diferentes vecindades durante el proceso. Al inicio, además de una solución de partida, se definen un conjunto de vecindarios, con los que se va a trabajar. Una vez se establezcan los vecindarios se procede a seleccionar un vecino de dicho vecindario, con el cuál se hace una búsqueda local. Cuando ésta finaliza, se compara la mejor solución obtenida con la que había inicialmente. Si la nueva solución mejora a la original, la sustituye y la nueva se convierte en la solución actual, volviendo a realizar el mismo proceso. Si no se produce mejora, se repite el proceso, pero usando la siguiente vecindad.

2.1.2 Metaheurísticas basadas en población

Estos métodos se difrencian de aquellos basados en trayectoria en que los basados en población parten de un conjunto de soluciones iniciales para cada iteración, mientras los de trayectoria partían de una única solución inicial. Algunas de las metaheurísticas basadas en población más extendidas (Sebastián Lozano, 2020) son:

- **Algoritmos genéticos (Genetic Algorithm GA):** son metaheurísticas basadas en población muy robusta y se basan en la teoría de la evolución. Consisten en crear secuencias, evaluarlas y reutilizar las mejores soluciones para generar descendencia. Para llevar a cabo estos algoritmos lo hacemos mediante tres fases: selección, reproducción y reemplazo. Primero se seleccionan las mejores soluciones para que pasen a la fase de reproducción. Después estas son modificadas mediante operadores de mutación y cruce. Por último, hay que decidir si entran en la población y, en caso de

que lo hagan, a que otra solución sustituye. Cuando se termina, se vuelve a hacer otra iteración y así sucesivamente hasta llegar al criterio de parada.

- **Scatter Search (SS)**: este método utiliza estrategias sistemáticas para conseguir soluciones que mejoren a las que había anteriormente y que pasen a la siguiente generación. Parte de una población inicia a la que se le aplica un método de mejora. Después se crean de forma sistemática subconjuntos a los que se aplican un operador de orden definido y se combinan para crear soluciones. A cada solución creada se le vuelve a aplicar un método de mejora y se actualiza el conjunto de soluciones sustituyéndo con las nuevas, siempre que superen a las existentes en calidad o diversidad. Dependiendo del momento en el que se realice la actualización se diferencia entre Scatter Search estática (actualizamos tras la combinación y mejora) y Scatter Search dinámica (actualizamos el conjunto cada vez que generamos una nueva solución).
- **Differential Evolution (DE)**: esta metaheurística se emplea para problemas de optimización global con codificación real. El método es similar al del algortimo genético. Se parte de una población inicial formada por vectores y obtenida de manera aleatoria, definiendo los valores máximos y mínimos que se pueden adoptar. Para pasar a la siguiente generación se comienza calculando vectores mutantes. Dependiendo de cómo se lleva a cabo la fase de mutación se distinguen los diferentes tipos de Differential Evolution que se denotan como DE/x/y/z. Una vez obtenido el vector mutante, se aplica un operador de cruce y se genera una nueva solución. Finalmente hay que seleccionar una de las dos (la obtenida de la mutación o la obtenida del cruce) para que pase a la siguiente generación. Sólo una puede pasar para poder mantener constante el tamaño de la población.

2.2 Variable Neighbourhood Search

En este capítulo se va a comentar la metaheurística que se usará para resolver el problema expuesto en el capítulo 2. El problema consistía en un entorno flowshop de permutación con minimización del total weighted completion time para dos conjuntos de trabajos. Para resolverlo, de entre todas las metaheurísticas posibles expuestas en el capítulo anterior, se va a utilizar la VNS (Búsqueda en Vecindad Variable). La metaheurística VNS se ha utilizado mucho en los últimos años para problemas de programación de la producción, como por ejemplo en Lei & Guo (2011, 2014) o en el artículo de Deming Lei “Variable neighborhood search for two-agent flow shop scheduling problem” el cuál se va a utilizar como base para este trabajo, ya que se seguirá el mismo método de resolución seguido por Lei (2015). Sin embargo, apenas se ha usado para programación multiobjetivo. Aunque ya se ha explicado un poco el algoritmo anteriormente, ahora se verá con más detalle.

Para comenzar, hay que crear una solución inicial o de partida que será además la mejor solución hasta que sea mejorada por alguna otra. Para la creación de la solución inicial se cuenta con muchas formas de generar esta solución, ya que puede hacerse de manera aleatoria, siguiendo alguna regla de despacho o mediante alguna otra heurística que nos puede proporcionar una solución. En este caso concreto, para calcular esta solución de partida, se va a usar una de las reglas básicas de despacho conocida como Weighted Shortest Processing Time first. Esta regla de despacho, cuyas siglas que vamos a usar de ahora en adelante son WSPT, consiste en procesar siempre en primer lugar el trabajo con mayor valor del cociente entre el peso y el tiempo de proceso. Por tanto, se partirá de una solución en la que los trabajos están ordenados de mayor a menor cociente entre peso y tiempo de proceso w_j / p_j.

Una vez tenemos la solución inicial (WSPT), trabajaremos con ella para buscar posibles soluciones que mejoren el valor de la función objetivo. Esta búsqueda hay que realizarla usando la metaheurística VNS, la cual es una familia de metaheurísticas basadas en vecindades que usan diferentes tipos de vecindades para la búsqueda del óptimo local. Estas vecindades hay que elegirlas antes de comenzar, ya que existen varios tipos de ellas muy diferentes unas de otras. Para el problema en cuestión, se van a usar tres vecindades (Lei, 2015) que se explicarán a continuación:

- **Intercambio (swap)**: este tipo consiste en el intercambio de dos trabajos cualesquiera que sean dentro de la secuencia. No tienen por qué ser adyacentes (esa sería otra vecindad conocida como adjacent swap). Una secuencia de n trabajos tiene $\frac{n(n-1)}{2}$ vecinos.

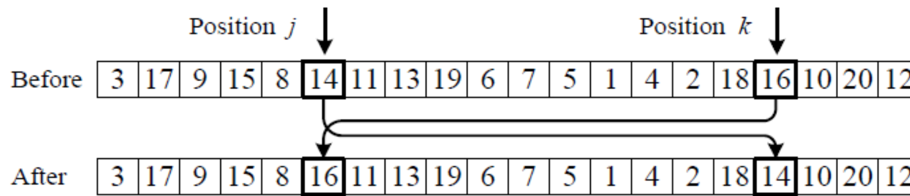

Figura 6: *Intercambio (Swap). Fuente: Diapositivas Programación y Control de la Producción (2020).*

- **Inserción (insertion):** esta vecindad consiste en extraer un elemento de la secuencia de *n* trabajos que tenemos e insertarlo en alguna de las posiciones posibles de la secuencia. Una secuencia de *n* trabajos tiene $(n-1)^2$ vecinos por inserción.

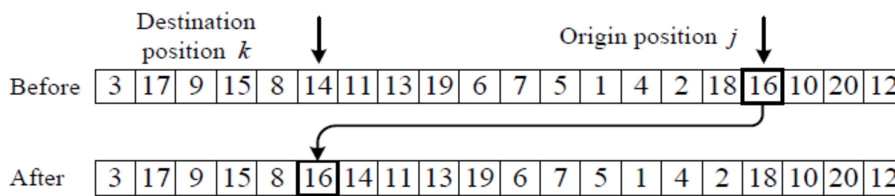

Figura 7: *Inserción (Insertion). Fuente: Diapositivas Programación y Control de la Producción (2020).*

- **Learner:** es más compleja que las anteriores. En primer lugar, copiamos y vaciamos la secuencia inicial *S* de *n* trabajos. Luego generamos aleatoriamente otra secuencia de la cual elegiremos, también aleatoriamente, algunos trabajos (entre *1* y *n/2*). Estos trabajos elegidos se insertan en la secuencia que S que teníamos vacía en las posiciones en los que estaban en la otra. Por último, los trabajos que aún no han sido insertados de la copia de S, se insertan también en S en el orden en el que estaban.

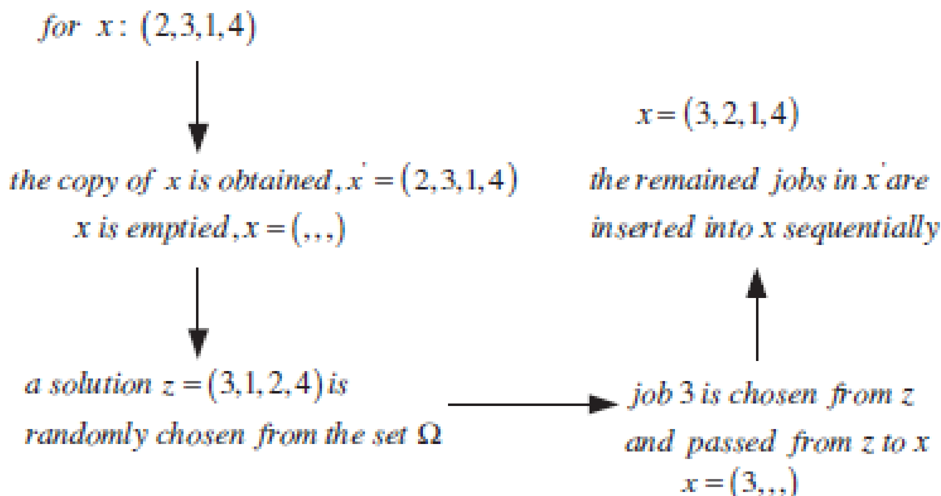

Figura 8*: Vecindad Learner. Fuente: Computers and Industrial Engineering, Deming Lei (2015).*

Teniendo en cuenta que estas son las tres vecindades que se van a usar para llevar a cabo la metaheurística, también hay que saber que existen diferentes variantes de Variable Neighbourhood Search. La VNS se puede realizar de varias formas, de hecho, cualquier metaheurística que de alguna forma utilice más de una vecindad se puede catalogar como VNS, por lo que a continuación se explicarán dos de las más sencillas (Lozano Segura, 2020).

2.2.1 Variable Neighbourhood Descent (VND)

La primera variante que se va a ver parte de una solución inicial y, definiendo los tres tipos de vecindades que se han visto anteriormente (V1, V2, V3), se aplica primero la vecindad V1 a la solución inicial. Hay que buscar en toda la vecindad al completo y pueden presentarse dos opciones:

- Se encuentra una solución mejor que la de partida. En este caso se toma la nueva solución como base y volvemos a aplicar V1.
- En toda la vecindad no se encuentra ninguna solución que mejore a la que había, entonces se aplica V2 a la solución actual.

Al aplicar V2 se vuelve a tener las dos mismas opciones, por lo que en caso de que algún vecino mejore la actual solución se tomaría como base y se volvería a aplicar V1 (primer paso). En caso contrario se seguiría aplicando el mismo método con la vecindad V3. Al terminar el proceso, se toma la solución que se haya obtenido como solución de partida y se vuelve a realizar otra vez el mismo proceso, iterando sucesivamente hasta alcanzar el criterio de terminación. Este método no garantiza optimalidad, pero sí que la solución obtenida sea óptimo local de varias vecindades diferentes al mismo tiempo.

2.2.2 Reduced Variable Neighbourhood Search (RVNS)

La otra variante, RVNS, también parte de una solución inicial y se definen los tres tipos de vecindades que se han visto anteriormente (V1, V2, V3). Luego se aplica primero la vecindad V1 a la solución inicial para obtener de forma aleatoria un único vecino, el cual se compara con la solución inicial y pueden presentarse dos opciones:

- El vecino obtenido aleatoriamente es mejor que la solución de partida. En este caso se toma como base y se vuelve a aplicar V1.
- El vecino aleatorio no mejora a la solución que había, entonces se aplica V2 a la solución actual para volver a obtener de forma aleatoria un vecino.

Al aplicar V2 se vuelven a tener las dos mismas opciones, por lo que en caso de que el vecino mejore la solución actual se tomaría como base y se volvería a aplicar V1 (primer paso). En caso contrario se seguiría aplicando el mismo método con la vecindad V3. Al terminar el proceso, se toma la solución que se haya obtenido como solución de partida y se volvería a realizar otra vez el mismo proceso, iterando sucesivamente hasta alcanzar el criterio de terminación (número máximo de iteraciones).

3 MÉTODO DE RESOLUCIÓN

El primer paso es establecer que algo es posible,
entonces la probabilidad ocurrirá.

-Elon Musk-

Para continuar, en este capítulo se va a proceder a describir todo lo que se refiere a la implementación del algoritmo explicado en el apartado anterior (capítulo 2) para la resolución del modelo de programación de la producción comentado en el capítulo 1. Dicho modelo estaba compuesto por dos conjuntos de trabajos y tenía como objetivo la minimización de la función objetivo (en este caso el total weighted completion time) de ambos conjuntos. Para resolver este problema se va a utilizar la metaheurística VNS. Para llevar a cabo la implementación se ha decidido utilizar el lenguaje de programación C. Hay que codificar el algoritmo VNS que se ha descrito anteriormente, codificando los tres tipos de vecindades que íbamos a usar para llevar a cabo el proceso. Además, hay que ser capaz de calcular la función objetivo de ambos conjuntos, ya que, como se explicó, hay que minimizar la del conjunto A, siendo la del B menor que el valor de épsilon (ε). Para poder comparar las soluciones que ofrece el programa se usará una batería de instancias con distintos números de trabajos y dos máquinas.

Para el desarrollo de los códigos se ha utilizado el compilador *Code::Blocks*, que es un entorno de desarrollo integado de código abierto y permite el uso de los lenguajes de programación C y C++. Dentro de este compilador, se ha usado la librería *Schedule* que ha sido aportada por el personal docente de la asignatura de "Programación y Control de la Producción", perteneciente al Grado en Ingeniería de Organización Industrial. Finalmente, una vez se han obtenido las soluciones a través de nuestro código, éstas han sido pasadas a Microsoft Excel para su estudio.

Aunque en este capítulo se van a explicar las partes más importantes del código que se ha generado, es posible encontrar el código completo al final del proyecto, en el apartado Anexo.

3.1 Cálculo de la Función Objetivo

En primer lugar, hay que recordar que la función objetivo que se está buscando minimizar es el total weighted completion time para ambos conjuntos. El cálculo de esta función objetivo (FO) es una de las partes más importante de este código, ya que esta será la base para que después se calcule de forma correcta la metaheurística. Por ello se debe recordar primero que el problema es biobjetivo, por lo que hay que calcular dos funciones objetivo, una para el conjunto A y otra para el conjunto B. Para el cálculo de las FO se utilizarán dos funciones en C en las que realizará el mismo proceso en ambas, pero al llegar al final una calculará la FO de A y la otra la de B.

Las funciones que se van a usar, denominadas *"FObjConjuntoA"* y *"FObjConjuntoB",* recibirán como parámetros el número de trabajos total y el número de trabajos de los conjuntos (que será la mitad), los valores

de los pesos para cada trabajo y del vector secuencia al que se quiere calcular la FO y por último un vector con los valores del completion time (tiempo de terminación) de cada trabajo. La función devolverá el valor de la FO obtenida para dicha secuencia. Dentro de estas funciones, se realizará el proceso de dividir el vector de tiempos de terminación en dos vectores, uno para cada conjunto separando así los elementos pertenecientes a cada conjunto. Se hará también lo mismo con los pesos, y una vez se haya hecho esto, solo quedará emplear la fórmula de la función objetivo. Para ello, se utilizará un bucle for para ir sumando los valores, ya que esta FO es un sumatorio $(\sum w_j C_j)$, como se muestra en el pseudocódigo que se puede ver en la Figura 9:

FObjConjuntoA

```
Input: instance data (WA, CA)
Output: FOA
begin
    for j=0 to M do
        FOA := FOA+WA(j)*CA (j);
    return FOA
```

Figura 9*: Pseudocódigo de la función de cálculo de la función objetivo de los trabajos del conjunto A.*

Para poder llegar a calcular estas funciones objetivo será necesario obtener dicho valor del tiempo de terminación, lo cual se conseguirá mediante otra función que se llamará *"completiontime"* y se encargará de calcular los completion time en función del número de máquinas que se tienen en cada instancia de entrada. Dicha función recibirá los valores del número total de trabajos *N*, el vector secuencia con el que se esté trabajando y la matriz de tiempos de proceso, los cuales son generados de manera aleatoria. Primero, lógicamente hay que definir e inicializar todas las variables y vectores que se van a usar para poder llevar a cabo esta función. En este caso, la función se hace como un puntero, ya que deberá devolver un vector con memoria dinámica. Dicho vector será el de los tiempos de terminación, y para ello se trabajará con un bucle for, dentro del cual se aplicará una fórmula u otra en función de si se cumple o no la condición marcada por el if. En la Figura 10, se muestra el cálculo de los tiempos de terminación:

Completiontime

```
Input: instance data (C1, p2)
Output: C2
begin
    for j=1 to N do
        if C2(j-1) <= C1(j) then
            C2(j) = C1(j) + p2(j);
        else
            C2(j) = C2(j-1) + p2(j);
        return C2
```

Figura 10*: Pseudocódigo de la función de cálculo del tiempo total de terminación.*

Una vez realizados estos cálculos, se incluirá el resultado del completion time en las funciones de cálculo de la FO que vimos anteriormente, y con estas se podrá calcular las FO de cualquier secuencia que se introduzca, lo cual será muy importante para llevar a cabo el proceso de resolución de la metaheurística.

3.2 Cálculo de las vecindades

Lo segundo que hay que hacer para poder después construir la metaheurística (VNS) será ser capaces de calcular los tres tipos de vecindades que se habían definido en el capítulo 2. Estas tres vecindades serán la base para la posterior realización de las metaheurísticas, por lo que es muy importante definirlas correctamente. Para ello, se han utilizado tres funciones diferentes, una para cada una de las vecindades que luego serán incluidas en el cáculo de la metaheurística, y ahora nos se explicará paso a paso cómo han sido calculadas.

3.2.1 General Swap

En primer lugar, se comenzará con la primera de ellas, que como ya se ha explicado se conoce como General Swap (Intercambio general). Es importante distinguirla de otro tipo de vecindad como es el Adjacent Swap (Intercambio adyacente). El adyacente se basa, como su propio nombre indica en intercambiar los valores de la secuencia que se tenga por aquellos que están a sus lados. Sin embargo, el General permite cambiar la posición de un valor de la secuencia por la de cualquier otro, por lo que se consiguen muchas más posibilidades y muchos más vecinos para calcular. Para este caso que se nos presenta se crea una función que se denominará *"v1_swap"* y recibirá como parámetros los valores de número de trabajos (*N*) y un vector secuencia. Esta secuencia que hay que pasarle a la función será, al principio, la secuencia inicial obtenida, y después con las iteraciones que se realicen se le pasará la que en ese momento constituya la mejor solución, es decir el vecino con mejor valor de la FO. En esta función, lo primero que se debe hacer es inicializar y definir los vectores y variables que se vayan a utilizar. En este caso concreto, debemos definir un vector con memoria dinámica (función *malloc*), que será el que devuelva la función al final. Este vector será el vecino que se conseguirá calcular con nuestra vecindad Swap. Además, para poder llegar a calcularlo se definirán dos números enteros a los que luego se les dará valores aleatorios desde 0 hasta el número de trabajos que haya en cada caso. Esto servirá para conseguir calcular un vecino por general swap de forma aleatoria, sin que se deba participar en el proceso de selección de las posiciones que serán intercambiadas. Para ello, como se puede ver continuación, se coge la secuencia que se había pasado como parámetro a la función y una copia de esta, y se intercambian las posiciones escogidas por los valores enteros aleatorios que habíamos calculado.

Para el cálculo de números aleatorios se utilizará la función *rand (),* una función de C que ya viene incluida en las librerías que se han usado y a la que solo hay que darle el valor máximo hasta el que puede llegar el número (en nuestro caso *N*), por lo que los números aleatorios se generan automáticamente. Hay que tener en cuenta que, si los valores aleatorios que se generen son iguales, hay que cambiarlos volviendo a generar dos valores nuevos, ya que, en caso de calcular el vecino con dos iguales, se quedaría la misma secuencia que había al comenzar.

V1_swap

Input: A, a1, a2

Output: resultado

begin

a1, a2 generate random

for j=0 to N do

resultado (j) = A(j);

if a1 != a2 then

resultado(a1) := A(a2);

resultado(a2) := A(a1);

return resultado

Figura 11*: Pseudocódigo de la función de cálculo de la vecindad General Swap.*

Hay que recordar siempre liberar la memoria del vector, sobre todo en los casos en los que se van a usar vectores con memoria dinámica. Esto se hace utilizando la función free () y poniendo dentro del paréntesis el vector que se quieran liberar.

3.2.2 Insertion

En segundo lugar, se tiene la segunda de las vecindades que se van a usar para el cálculo de nuestra metaheurística. Se trata de la conocida como Insertion (Inserción) y consiste en seleccionar uno de los elementos de la secuencia que haya en el momento de su cálculo y extraerlo de su posición, insertándolo después en otra posición aleatoria. A diferencia del caso anterior, la secuencia a la que se le calculará la inserción será, en teoría siempre, al mejor vecino que haya en cada iteración. Sin embargo, se puede dar el caso de que la que llegue sea la secuencia inicial, ya que es posible que el vecino calculado en la primera vecindad no la haya mejorado. Para este caso, al igual que para la vecindad anterior, se crea una función llamada *"v2_insert"* que recibirá como parámetros los valores de número de trabajos (*N*) y un vector secuencia. Para esta función, lo primero que se va a hacer es inicializar y definir los vectores y variables que se vayan a utilizar. En primer lugar, al igual que se hizo en la primera vecindad, hay que definir un vector con memoria dinámica (función *malloc*) que será el que la función devolverá al final como resultado. Este vector será el vecino que se conseguirá calcular con nuestra vecindad Insertion. Para poder llegar a calcularlo, primero se realizará una copia de la secuencia introducida como parámetro en este vector resultado y se definirán dos números enteros a los que luego habrá que dar valores aleatorios desde 0 hasta el número de trabajos que haya en cada caso (*N*). Esto servirá para calcular los vecinos de forma aleatoria, sin que haya que intervenir introduciendo los valores que se van a usar.

Para el cálculo de números aleatorios, como en el caso anterior, se utiliza la función *rand ().* Además de esta, habrá que usar otras tres funciones que están incluidas en la librería *Schedule*, la cuál se está usando. La primera de ellas será *search_vector ()* y se utilizará para buscar en la secuencia el primer valor aleatorio que se haya calculado. Esta función devuelve la posición de dicho valor, por lo que se guardará en una variable entera. La segunda función que se va a usar es *extract_vector (),* que se encargará de extraer de la secuencia resultado el valor de la posición que se había guardado en una variable. Para terminar, está la última, que será *insert_vector (),* y lo que hará será insertar el valor que marca el número que se había extraído como posición de la secuencia inicial, en la posición de la secuencia resultado que se obtenga del segundo número aleatorio que se haya calculado. Se puede ver lo explicado en el siguiente pseudocódigo:

V2_insert

Input: B, a, random1, random2

Output: resultado

begin

random1, random2 generate random

a := random1 position in vector resultado;

Extract value a from vector resultado;

Insert value B(a) in position random2 from vector resultado;

return resultado

Figura 12*: Pseudocódigo de la función de cálculo de la vecindad Insertion.*

Con esto se habría calculado la segunda de las tres vecindades que hay que usar para el cálculo de la metaheurística, por lo que queda ver la última de ellas.

3.2.3 Learner

Para terminar, hay que ver la última y más compleja de las vecindades que se van a utilizar, denotada como Learner en el artículo *Variable neighborhood search for two-agent flow shop scheduling problem, Lei (2015).* Como en el caso anterior, la secuencia a la que se le calculará la inserción será al mejor vecino que haya en cada iteración, a no ser que el vecino calculado en las primeras vecindades no haya mejorado a la solución inicial, en cuyo caso llegará esta misma. Para este caso, al igual que para las otras vecindades, se crea una función llamada *"v3_learner"* que recibirá como parámetros los valores de número de trabajos (*N*) y un vector secuencia. Lo primero que hay que hacer es, como en las anteriores, inicializar y definir los vectores y variables que se van a utilizar. En primer lugar, al igual que se hizo en las vecindades anteriores, hay que definir un vector con memoria dinámica (*malloc*) que será el que la función devolverá al final como resultado. Este vector será el vecino que se conseguirá calcular con la vecindad Learner. Además de este, también se definirá de la forma habitual otros vectores. Para uno de ellos se generará una secuencia aleatoria y se le realizará una copia. Otro servirá como vector que habrá que rellenar y con el que se trabajará. Una vez todo esté definido, habrá que calcular un número aleatorio desde 1 hasta *N*/2 (la mitad del número de trabajos que haya) que será el número de trabajos que se van a coger de la secuencia aleatoria para formar la solución. Después se generan tantos números aleatorios como indicara el número generado anteriormente. Estos últimos serán los trabajos que, de forma parecida a lo visto en Insertion, se extraerán de la secuencia aleatoria y se insertarán en la secuencia que había que rellenar en sus mismas posiciones. La forma de hacer este proceso es la que se puede observar en la Figura 13, mediante un bucle for para que se realice tantas veces como indique el número aleatorio calculado al principio. Una vez que se tienen estos valores insertados en la secuencia que será la solución, ya sólo quedaría un paso. Esta vez habrá que coger los trabajos que falten por insertar en la secuencia solución y buscarlos e insertarlos en el orden en el que estuvieran en la secuencia inicial que se pasó como parámetro, pero en las posiciones que falten por rellenar. Todos estos procesos se harán también con las funciones de C search_vector (), extract_vector () e insert_vector (), como se puede ver en la Figura 13:

V3_learner

Input: C, sec, aleatorio, L

Output: resultado

begin

aleat generate random

for i=0 to aleat do

c1 := aleatorio (i) position in vector sec;

c2 := aleatorio (i) position in vector C;

Extract value c2 from vector C;

Extract value c1 from vector L;

Insert value aleatorio(i) in position c1 from vector L;

j :=0;

for i=0 to N do

if sec(i) != L(i) then

Extract value i from vector L;

```
                Insert value C(j) in position i from vector L;
                j := j+1;
    for j=0 to N do
        L(j) = resultado(j);
    return resultado
```

Figura 13*: Pseudocódigo de la función de cálculo de la vecindad Learner.*

En conclusión, estas tres vecindades han sido definidas en tres funciones para después poder trabajar con ellas más fácilmente. Con esto, se puede empezar a explicar el proceso de generación de nuestra metaheurística.

3.3 Variable Neighbourhood Search

3.3.1 Solución inicial

Como se expuso en capítulos anteriores, toda esta metaheurística debe partir de una solución inicial. Esta solución de partida puede ser calculada de muchas formas, ya sea por ejemplo aleatoriamente o mediante alguna regla de despacho. En este caso, se usará una regla de despacho que será la Weighted Shortest Processing Time (WSPT). Esta regla consiste en calcular el cociente entre el peso de cada trabajo y la suma de los tiempos de proceso de cada trabajo en todas las máquinas. Este cálculo no es muy complicado, pero hay que tener en cuenta los dos conjuntos de trabajo. Para realizar este método se usa una función de C que se encuentra en nuestra librería *Schedule* llamada sort_vector (), aunque antes de eso hay que realizar el cálculo de wj / Cj para poder después ordenarlos de mayor a menor. A esta función de ordenar el vector hay que introducirle la secuencia que se quiera ordenar y su tamaño, además de si hay que ordenarlo de manera ascendente o descendente, en este caso descendente, como se puede ver en la Figura 14.

Input: vA, vB, w_j , pA_j , pB_j

Output: WSPT

begin

```
    for j=0 to N/2 do
        vA(j) = w(j)/pA(j);
    j := N/2;
    for k=0 to N/2 do
        vB(j) := 0;
        if j < N then
            vB(j) = w(j)/pB(j);
            j := j+1;
    Sort vector vA in decreasing order in WSPTA vector;
    Sort vector vB in decreasing order in WSPTB vector;
    for j=0 to N/2 do
        WSPT(j)=WSPTB(j)
```

```
j := N/2;
for k=0 to N/2 do
    if j < N then
        WSPT(j)=WSPTA(j)
        j := j+1;
return WSPT
```

***Figura 14**: Pseudocódigo del cálculo de la solución inicial.*

3.3.2 VNS

En este punto se va a reunir todo lo explicado anteriormente para llevar a cabo el cálculo de la metaheurística. Para ello, se hará uso de las funciones objetivo calculadas en el apartado 1 y de las vecindades del apartado 2.

En primer lugar, hay que definir todas las variables que se vayan a utilizar. Una vez hecho esto, partiendo de la secuencia que se tomará como solución inicial, se calculará su función objetivo para los conjuntos A y B. Estas funciones objetivo de la solución inicial serán muy importantes, ya que la del conjunto A se tomará como mejor valor actual y será la que se tratará de minimizar con cada vecino que se vaya calculando. Por otro lado, tendremos el valor de épsilon. Este valor también es muy importante para el desarrollo de la metaheurística, ya que toda solución calculada cuyo valor de la FO de su conjunto B sea mayor que épsilon, será directamente desechada. Una vez se hayan calculado las FO de la solución inicial y se haya establecido el valor de épsilon, ya si que se empezará el método de Variable Neighbourhood Search. Para ello, se aplica en primer lugar la función de cálculo de la vecindad Swap y se almacena en un vector. Este vector será por tanto el vecino calculado por Intercambio de la solución inicial. Una vez se tenga el vecino, solo habrá que volver a aplicar las FO de ambos conjuntos para calcular las de dicho vecino. Cuando se obtienen, llega el momento decisivo, en el que hay que observar si la FO del conjunto B del vecino calculado es menor que el valor de épsilon. En caso de no lo sea, se pasa al siguiente paso, ya que este vecino es infactible. Si la FO de B si fuese menor que ε, se pasaría a comparar la FO del conjunto A del vecino con la que se había guardado en la variable *mejor*. En caso de que también sea menor, se toma este vecino como la mejor solución y como base para volver a realizar el mismo proceso con la vecindad Swap. Una vez se calcule un vecino que no mejore a la solución actual (guardada en el vector *vecino*), se pasaría a la segunda vecindad. Esta segunda vecindad (Insertion), al igual que la última de ellas (Learner) se llevarían a cabo exactamente igual. Esto quiere decir calculando primero el vecino mediante la función correspondiente a cada vecindad y luego calculando las FO de ambos conjuntos y observando si minimizan los valores actuales y de épsilon. En la Figura 15 se puede observar el pseudocódigo de la metaheurística.

Un detalle que hay que tener en cuenta es que, cada vez que se calcule un vecino que mejore a la solución actual, el proceso debe volver a empezar con esa nueva solución desde el principio, es decir, volver a empezar con la primera vecindad. Por tanto, si el proceso se encuentra en V3 y se consigue una mejora se volvería a V1 y de ahí se pasaría a V2 cuando no mejorase y luego a V3 de nuevo. Por tanto, una vez se llega a V2, dentro del segundo if en el que se compara la FO de A con *mejor*, hay que meter de nuevo el proceso de generación de la vecindad V1. En el caso de V3 hay que volver a meter V1 y dentro de esa V1 se metería otra vez V2, para así poder completar el bucle tal como pide la metaheurística VNS.

```
Input: instance data
Output: Best solution vecino and best objective function value mejor
Begin
    Calculate initial solution I;
```

```
        Set best known solution vecino := I;
        Calculate best objective function value so far mejor := Obj(I);
        improving := TRUE;
        while improving := TRUE do
            improving := FALSE;
            foreach neighbourhood V do
                x :=  neighbour in V using v1_swap, v2_insert, v3_learner;
                FOA := FobjConjuntoA (x);
                FOB := FobjConjuntoB (x);
                if FOB <= epsilon then
                    if FOA1 <= mejor then
                        mejor := FOA1;
                        vecino := x;
                        improving := TRUE;
                        return to V1;
    return Best solution vecino and best objective function mejor
```

Figura 15*: Pseudocódigo del cálculo de la metaheurística VNS.*

Todo este proceso es iterativo, por lo que cuando acaba se vuelve a empezar desde el principio, y así sucesivamente hasta llegar a la condición de parada. Dicha condición está especificada como un bucle while que incluye toda la metaheurística. En este caso, se ha tomado como criterio de parada el número máximo de iteraciones (n_iter = 100000).

Por último, se puede comentar que se ha creado una última función denominada *"output"* que ayudará a almacenar los datos de las soluciones en un fichero, con el objetivo de exportarlo posteriormente a un archivo de Excel para su estudio. Para ella se han usado dos variables char* que serán las que después se introduzcan como *argv [1]* y *argv [2].* Después aparecen los valores que se escribirán en el fichero de texto de salida que serán la secuencia solución, y el valor de su función objetivo, así como el valor de épsilon.

4 ANÁLISIS COMPUTACIONAL

La producción de demasiadas cosas útiles da como resultado demasiadas personas inútiles.

- Karl Marx-

En este capítulo se va a proceder a analizar los datos obtenidos tras la resolución de una batería de instancias con el método propuesto anteriormente. Para ello, se analizará la metaheurística usada en función del valor de las funciones objetivo que se van a obtener para cada instancia. Estas instancias forman una batería con la que se va a trabajar, y están formadas por grupos de diez instancias en función del número de trabajos (*N*) y número de máquinas (*M*) que tiene cada una de ellas. Las que tienen los mismos números de trabajos y máquinas se compararán entre ellas.

Para el estudio de la función objetivo se va a utilizar un nuevo concepto que será el factor RDI. Estas siglas pertenecen en inglés a Relative Deviation Index, o Índice de Desviación Relativa. Este índice permitirá realizar una mejor evaluación de las soluciones obtenidas gracias al código y la metaheurística implementada. El RDI se calculará mediante la siguiente fórmula:

$$RDI = \frac{\text{FO} - \text{FObest}}{\text{FOw} - \text{FObest}} * 100$$

En esta fórmula, FO hace referencia a la función objetivo del conjunto de trabajos A, que en el caso que se está estudiando es el total weighted completion time, de la secuencia que se quiere evaluar, mientras que FObest y FOw son el mejor y el peor valor de la función objetivo de A que se han alcanzado en cada instancia respectivamente. Por lo tanto, cuando el valor RDI=0 se debe a que dicho valor de la función objetivo es el mejor, mientras que cuando RDI=1 el valor de la función objetivo es el peor de los obtenidos en la instancia. Para trabajar con este índice se usará el factor ARDI, que consiste en el valor medio de los RDI que se hayan calculado. Este se calculará por tanto como la media de los RDI que se hayan obtenido.

4.1 Generación de instancias

Para poder desarrollar el modelo y llegar a resolverlo, hay que usar unos problemas, los cuáles serán generados haciendo uso de otro código en C para la generación de instancias. Se van a considerar 6 problemas en función del número de trabajos y máquinas que tendrá cada uno. Para cada uno de estos problemas se generarán 10 instancias, por lo que en total habrá 60 instancias. Para cada instancia se generarán un número de trabajos, una matriz de tiempos de proceso para cada trabajo y sus pesos, así como el valor de épsilon (ε) que se usará para resolver el problema. Este valor de ε se calculará como la función objetivo del conjunto B de la secuencia obtenida mediante el uso de la regla WSPT, y estableciendo los trabajos del conjunto A en primer lugar ordenados según WSPT y los del B al final de la secuencia, también ordenados por WSPT. Tanto los tiempos

de proceso y los pesos serán generados de manera aleatoria, tomando valores entre 1 y 100. Además de estos parámetros, se usarán los valores de delta (δ) que irán de 0 a 0,5 con el objetivo de ir disminuyendo el valor de épsilon (ε= ε ·(1-δ)).

Las instancias consistirán en archivos .txt los cuáles se pasarán al programa, que se encargará de cargarlos haciendo uso de la función de lectura *LecturaInstancias* que se ha desarrollado. Para esta función, hay que hacer uso de un valor que se ha llamado *argv[1]*. Este valor será el archivo de entrada, que en cada caso consistirá en una *instanciaX_2xn.txt*, siendo *X* y *n* el número de trabajos el número de la instancia.

4.2 Análisis de resultados

En la Tabla 1, se puede observar un ejemplo del problema generado para una instancia. En este caso sería para 2 máquinas y 10 trabajos, y en la tabla aparecen los valores de δ y ε (en función de δ: ε= ε ·(1-δ)), además de los valores de la FO para cada instancia que hay que recordar que consiste en el total weighted completion time del conjunto A y su índice RDI.

Instancias	δ	ε	FO	RDI
instancia4_2x10	0.00	126847.00	58642.00	0.00
	0.10	114162.30	59378.00	0.01
	0.20	91329.84	63104.00	0.07
	0.30	63930.89	77500.00	0.28
	0.40	38358.53	112738.00	0.80
	0.50	19179.27	125974.00	1.00

***Tabla 1**: Problema generado para n=10 y m=2.*

Como se puede observar en la Tabla 1, los valores que se le dan a delta hacen que el valor de épsilon vaya reduciéndose. Al ir reduciéndose el valor de ε, obtenemos unos valores de total weighted completion time del conjunto A cada vez mayores. Al principio, tenemos que los trabajos de B están más a la derecha en la secuencia y los de A se encuentran al principio. El aumento de valor de la FO se debe a que, a medida que delta aumente (ε disminuye) la factibilidad será más complicada, por lo que los trabajos de B tendrán que ir más a la izquierda y los de A se trasladarán más a la derecha.

Por otro lado, la Tabla 1 también se puede observar el valor del factor RDI para esta secuencia. En el primer caso con la ε más grande (δ =0), se alcanza el mejor valor de la FO de A para esta instancia, y será así para todas las instancias, ya que al ir disminuyendo ε se restringen las posibilidades de encontrar soluciones factibles y esto provoca el aumento del valor de la FO del conjunto A. En este primer caso, el valor del RDI es igual a 0, lo que quiere decir que es la mejor solución de esta instancia. En el otro extremo se encuentra el valor de RDI=1 cuando δ vale 0,5. Esto indica que es el peor valor de la función objetivo final que alcanza esta instancia. A medida que aumenta el valor de delta se puede ver como el RDI aumenta también.

Una vez se han obtenido los resultados de todos los problemas, es decir, de las 60 instancias se va a calcular el ARDI (RDI medio) de cada problema (para cada tamaño nxm) en función del valor de delta. En la Tabla 2 se muestran los valores obtenidos en este cálculo. Como se puede observar, los valores del RDI crecen según crece el valor de delta, tal como se vio también el la Tabla 1.

	δ					
Tamaño (mxn)	0	0.1	0.2	0.3	0.4	0.5
2x10	0.0000	0.1289	0.2484	0.5541	0.9368	1.0000
2x20	0.0000	0.0578	0.2356	0.6860	0.9749	1.0000
2x50	0.0000	0.0413	0.1593	0.3891	0.9091	1.0000
2x70	0.0000	0.0375	0.1506	0.3748	0.9248	1.0000
2x100	0.0000	0.0545	0.1676	0.4337	0.9275	1.0000
2x200	0.0000	0.0485	0.1828	0.4227	1.0000	1.0000

***Tabla 2**: ARDI de cada tamaño para cada delta.*

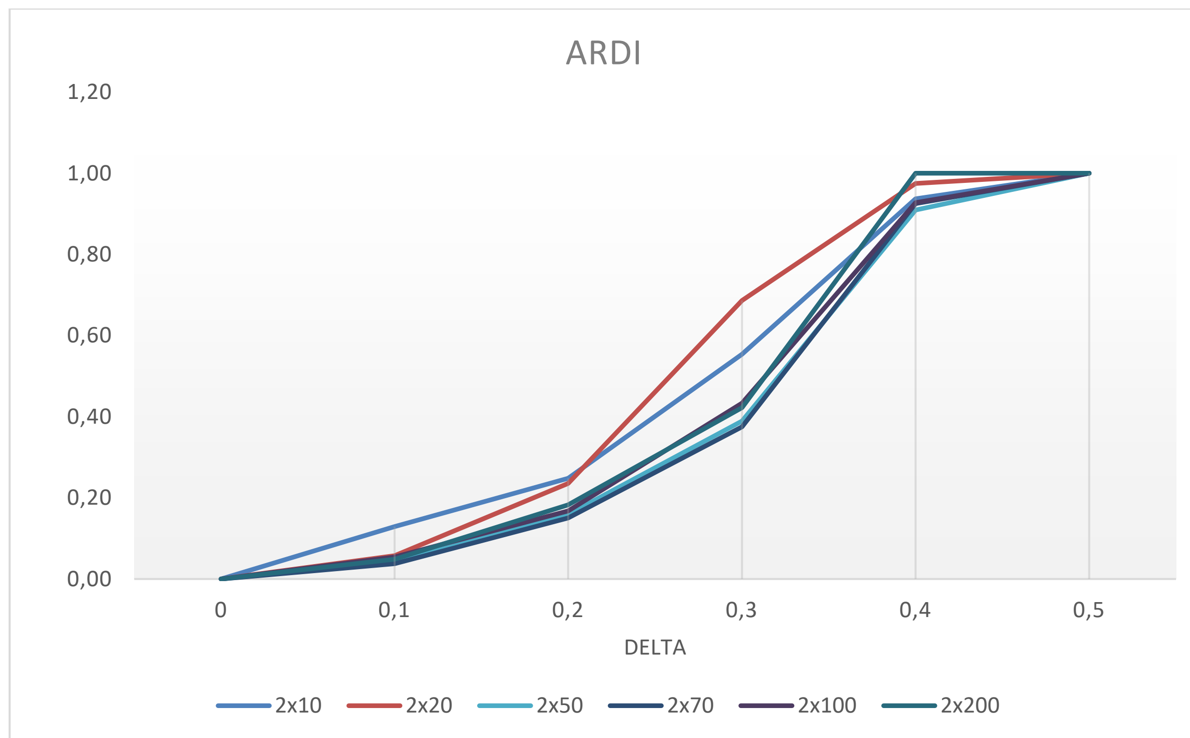

***Figura 16:** ARDI de cada tamaño para cada delta*

Como se puede apreciar en la Figura 16, conforme el valor de delta va aumentando, los valores del ARDI también aumentan. Esto es debido a que el aumento de delta y la consiguiente disminución de ε, provocan que los valores de la función objetivo del conjunto A de la secuencia solución sea cada vez mayor. Por ello, al empeorar el valor del total weighted completion time, el ARDI va aumentando hasta llegar a 1 en el peor de los casos, es decir, cuando δ =0,5.

Otra forma de reflejar estos datos es como se aprecia en la Figura 17, invirtiendo los datos de la gráfica. En esta se pueden apreciar perfectamente los valores para δ =0 y δ =0,5 como límites, ya que hacen el ARDI=0 y ARDI=1 respectivamente.

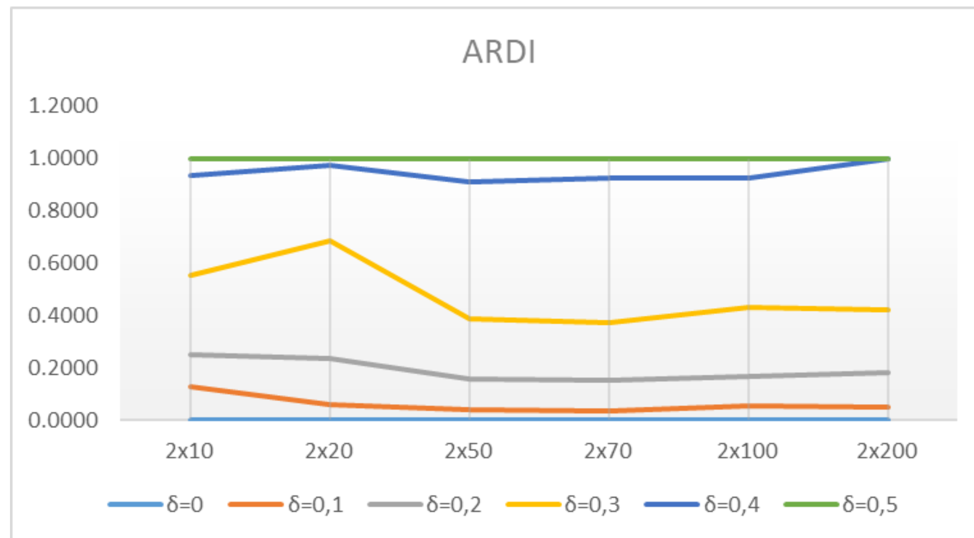

Figura 17: *ARDI de cada delta para cada tamaño*

A partir de ahora se van a analizar los diferentes valores de la función objetivo (total weighted completion time) que hemos obtenido para cada problema, es decir, para cada grupo de diez instancias dependiendo del número de trabajos y valores de delta. Para comenzar, vamos a estudiar los valores medios del factor RDI para las soluciones que hemos obtenido basándonos en el número de trabajos. Para ello tenemos la Tabla 3 que se muestra a continuación:

num trabajos	ARDI
10	0.48
20	0.49
50	0.42
70	0.41
100	0.43
200	0.44

Tabla 3*: ARDI en función del número de tabajos.*

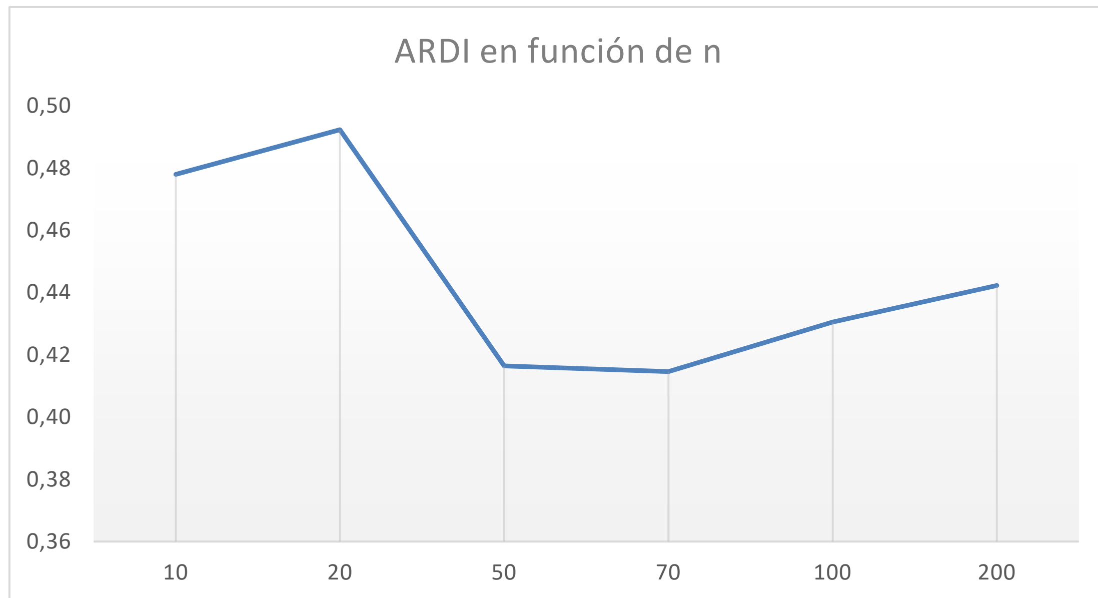

Figura 18: *ARDI en función del número de tabajos.*

De los valores de la Figura 18 se puede extraer, por tanto, que este algoritmo ofrece mejores soluciones para un mayor número de trabajos, ya que para los dos primeros tamaños calculados n=10 y n=20 el valor del factor ARDI es considerablemente mayor que para el resto de los problemas estudiados. Sin embargo, a medida que se aumenta el número de trabajos, los valores obtenidos son menores y similares.

A continuación, se pueden seguir estudiando los resultados del factor ARDI, clasificando los problemas en función del valor de delta que tengamos. En la Tabla 4 se puede ver la variación del índice ARDI en función del valor de delta (δ), es decir, según los diferentes valores que se hayan obtenido de épsilon a partir de la FO del conjunto B su disminución aplicándole la fórmula establecida (ε= ε ·(1-δ)).

delta	ARDI
0	0.00
0.1	0.06
0.2	0.18
0.3	0.46
0.4	0.95
0.5	1.00

Tabla 4*: ARDI en función del valor delta.*

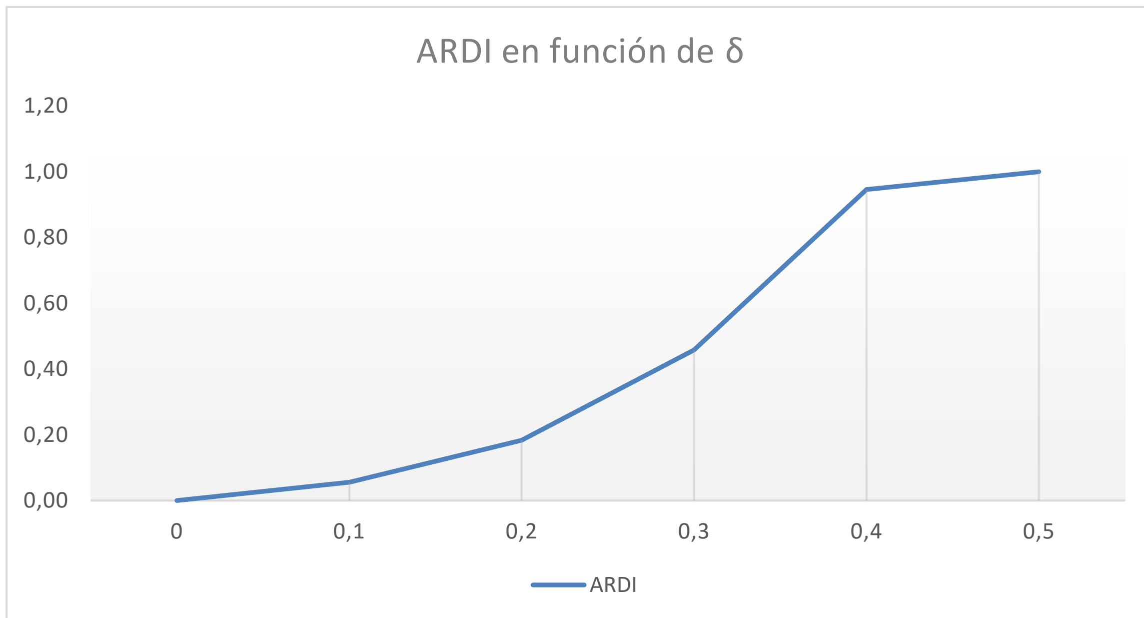

Figura 19: *ARDI en función del valor delta.*

En esta Figura 19 se puede apreciar que, como se explicó en la primera gráfica, las mejores soluciones se alcanzan para el valor de δ=0. Cuando se tiene este valor, el valor de épsilon es el original, hallado a partir de la función objetivo del conjunto B. Sin embargo, cuando se empieza a aumentar el valor de delta, el valor de épsilon disminuye, por lo que se restringen las posibles soluciones estableciendo un upper bound (límite superior) más pequeño. Esto hace que las soluciones que se encuentren sean peores que cuando δ=0, llegando a la peor solución estudiada cuando δ=0,5.

Por tanto, se ha presentado el ARDI de cada problema (grupo de instancias según su tamaño) para cada delta, así como los resultados de dicho índice para cada valor del número de trabajos y delta.

5 CONCLUSIONES

Aquel que lo intentó y no lo consiguió es superior al que ni lo intentó.

- Arquímedes-

En la actualidad la optimización de procesos y la programación de la producción están adquiriendo una importancia mucho mayor de la que habían tenido a lo largo de la historia. En los procesos de fabricación, por ejemplo, la optimización cobra mucha importancia dentro de una empresa, ya que debemos llevar a cabo dicha producción de una forma lo más eficiente posible para intentar conseguir los objetivos de minimización de costes y maximización de la calidad y los volúmenes de producción.

Este trabajo de fin de grado se ha centrado en el estudio de un problema de flowshop de permutación con dos conjuntos de trabajos y objetivo de minimización del total weighted completion time. Este es un problema sobre el que no se ha escrito mucho, ya que llegar a buenas soluciones con dos conjuntos de trabajos es complicado debido al problema multi objetivo y a la dificultad de minimizar los objetivos de ambos conjuntos. Para llevar a cabo la resolución de dicho problema se ha usado la metaheurística llamada Variable Neighbourhood Search.

El trabajo sigue una estructura que está compuesta para comenzar de un capítulo introductorio teórico en el que se expone la teoría de la programación y control de la producción, así como su notación habitual con el objetivo de aclarar todas las herramientas que usaremos posteriormente. A continuación en este capítulo teórico se procede a realizar una descripción completa del problema que se va a tratar de resolver y sus características principales.

Una vez finalizado el capítulo 1, continúa con el apartado de las metaheurísticas. En él, se explican los diferentes tipos de metaherísticas que hay, así como las metaheurísticas más importantes de cada tipo. Tras esta breve presentación se procede a explicar con mayor detalle la metaheurística VNS, la cuál se usará para resolver el problema de programación de la producción presentado en este trabajo.

Después de haber comentado y explicado paso a paso todas las herramientas que se van a usar para realizar y resolver nuestro modelo, llegamos al capítulo 3, en el que se verá y se explicará con detalle el código que se ha generado para llevar a cabo la resolución del problema. Se analizará cada parte del código y se explicará cómo se ha realizado y qué función tiene cada parte.

Para finalizar, está el capítulo 4, en el cuál se ha utilizado el código para llegar a unas secuencias y sus funciones objetivo solución, las cuáles se han usado, a través de la herramienta Microsoft Excel, para realizar un análisis de todos los resultados que ha aportado la metaheurística utilizada. Para ello se usa una serie de instancias, divididas por números de trabajos y de máquinas. En conclusión, el desarrollo de este trabajo, del modelo y de la metaheurística de resolución ha sido un trabajo complicado y ha requerido de muchas modificaciones, búsqueda y pruebas para llevar a cabo todo el problema y su resolución. Con todos los datos obtenidos a través del estudio y el análisis de las soluciones se puede concluir que se han hallado soluciones factibles a través del estudio de las vecindades de una secuencia de partida, y estas soluciones son peores para números de trabajos más pequeños y mejores para los más grandes. Además de esto, como ya se ha explicado,

las mejores soluciones se obtienen para los valores de δ=0 en todos los casos estudiados, ya que es en ese momento cuando el valor de épsilon es mayor. A partir de aquí, éste va disminuyendo, dejando menos posibilidad a secuencias factibles, por lo que los valores del total weighted completion time del conjunto A van empeorando a medida que épsilon disminuye. Esto se debe a que, aunque al principio tenemos que los trabajos de B están más a la derecha en la secuencia y los de A más a la izquierda, a medida que delta aumente (épsilon disminuye) la factibilidad será más difícil, por lo que los trabajos de B tendrán que desplazarse más a la izquierda y los de A más a la derecha, aumentando así el valor de la FO de A.

Con este objetivo que se ha propuesto (total weighted completion time), si cada conjunto de trabajos pertenece a unos clientes diferentes y cada trabajo tiene una prioridad (peso) distinta, se pretende minimizar el tiempo medio de terminación de los trabajos de cada conjunto teniendo en cuenta dichas prioridades.

Como se ha dicho, cada conjunto de trabajos pertenecerá a un agente, y ambos agentes competirán en el uso de unos recursos comunes en el proceso de fabricación. Por tanto, la meta será encontrar un programa que minimice una combinación de los objetivos de ambos agentes, que dependen del tiempo de finalización de sus trabajos. Poniendo como ejemplo una empresa que fabrique productos para dos clientes diferentes, deberá tratar de satisfacer los objetivos de ambos clientes. Con el método expuesto, se intentan satisfacer estos objetivos. Sin embargo, debido a la dificultad de minimizar ambos objetivos a la vez, se tomará como principal a uno de estos clientes (en el caso estudiado es el conjunto A), minimizando sus tiempos de fabricación, mientras se establecen unos límites de tiempo que los objetivos del segundo cliente (conjunto B) no podrán sobrepasar. De esta forma, siempre uno de los dos clientes, cuyos objetivos son los que se minimizarán, obtendrá unos resultados más beneficiosos.

6 REFERENCIAS

Lei, D. (2015). *Variable neighborhood search for two-agent flow shop scheduling problem.*

Ahmadi-Darania, M. H., Moslehia, G. & and Reisi-Nafchia, M. (2018). *A two-agent scheduling problem in a two-machine flowshop.*

Pérez González, P., Fernández-Viagas, V. & Framiñán, J. M. (2020). *Programación y Control de la Producción.*

Ballesteros Silva, P. P., Ballesteros Riveros, D. P. & Bravo Bolívar, J. E. (2013). *Application a constructive heuristic sequential programming for assigning multiple jobs to multiple machines in parallel.*

Pinedo, M. L. (2012). *Scheduling: Theoty, Algotithms, and Systems*.

Lozano Segura, S. (2020). *Métodos de optimización*

Hansen, P., Mladenovic, N. & Moreno Pérez, J. A. (2003). *Variable Neighbourhood Search. Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial. (No.19).*

Osorio Muriel, A. F., Brailsford, S. & Smith, H. (2014). *Un modelo de optimización bi-objetivo para la selección de tecnología y asignación de donantes en la cadena de suministro de sangre. Revista S&T, 12(30).*

Laumanns, M., Thiele, L. & Zitzler, E. (2006). *An efficient, adaptive parameter variation scheme for metaheuristics based on the epsilon-constraint method. European Journal of Operational Research (169).*

Pérez González, P., Fernández-Viagas, V. & Framiñán, J. M. (2020). *Permutation flowshop scheduling with periodic maintenance and makespan objective*.

Mavrotas, G. (2009). *Effective implementation of the e-constraint method in Multi-Objective Mathematical Programming problems.*

Chicano García, J. F. (2007). *Tesis doctoral: Metaheurísticas e Ingeniería del Software.*

Alancay, N., Villagra, S. & Villagra. A. (2014). *Metaheurísticas de trayectoria y poblacional aplicadas a problemas de optimización combinatoria.*

7 ANEXO

Para concluir, en este último apartado se incluye el código en C completo que se ha programado para el estudio del problema de flowshop de permutación de dos máquinas con dos conjuntos de trabajos para el total weighted completion time. Además, se incluye también el código elaborado para la generación de las instancias que posteriormente has sido resueltas y estudiadas.

7.1 Código en c

```
#include <schedule.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int *processtime(int N, int M, MAT_INT PT);
double FObjConjuntoA (int N, int M, VECTOR_DOUBLE Wj, VECTOR_INT orden, VECTOR_INT Ctime);
double FObjConjuntoB (int N, int M, VECTOR_DOUBLE Wj, VECTOR_INT orden, VECTOR_INT Ctime);
int *v1_swap(int N, VECTOR_INT A);
int *v2_insert(int N, VECTOR_INT B);
int *v3_learner(int N, VECTOR_INT C);
void output(char*entrada,char*salida,int N,VECTOR_INT secuenciafinal,double FO_final, double epsilon);
int *completiontime(int N, VECTOR_INT orden, MAT_INT PT);
MAT_INT LecturaInstancias(char *filename, int *jobs, int *machs, double *epsilon,  int print_flag);
VECTOR_DOUBLE LecturaInstancias2(char *filename, int *jobs, int *machs, double *epsilon, int print_flag);

int main(int argc, char *argv[])
{

    int n;
    int n_a;
    int m;
```

```
int j, k, i;
double eps;

srand(time(NULL));

LecturaInstancias(argv[1],&n,&m,&eps,YES);
LecturaInstancias2(argv[1],&n,&m,&eps,NO);

MAT_INT tiempos_proceso = DIM_MAT_INT(m,n);
tiempos_proceso = LecturaInstancias(argv[1],&n,&m,&eps,NO);
VECTOR_DOUBLE w = DIM_VECTOR_DOUBLE(n);
w = LecturaInstancias2(argv[1],&n,&m,&eps,NO);
n_a = n/2;

VECTOR_INT suma_p = DIM_VECTOR_INT(n);
suma_p = processtime(n, m, tiempos_proceso);

VECTOR_INT p_a = DIM_VECTOR_INT(n_a);
VECTOR_INT p_b = DIM_VECTOR_INT(n_a);

for(j=0; j<n_a; j++)
{
   p_a[j]=suma_p[j];
}
j=n_a;
for(k=0; k<n_a; k++)
{
   p_b[k]=0;
   if(j<n)
   {
      p_b[k] = suma_p[j];
      j++;
   }
}

//CREAR SECUENCIA DE PARTIDA WSPT
```

```
VECTOR_DOUBLE v = DIM_VECTOR_DOUBLE(n);
VECTOR_DOUBLE v_a = DIM_VECTOR_DOUBLE(n_a);
VECTOR_DOUBLE v_b = DIM_VECTOR_DOUBLE(n_a);
for(j=0; j<n_a; j++)
{
   v_a[j]=w[j]/p_a[j];
}
j=n_a;
for(k=0; k<n_a; k++)
{
   v_b[k]=0;
   if(j<n)
   {
      v_b[k] = w[j]/p_b[k];
      j++;
   }
}
for(j=0; j<n; j++)
{
   v[j]=w[j]/suma_p[j];
}

VECTOR_INT WSPT = DIM_VECTOR_INT(n);
VECTOR_INT WSPT_a = DIM_VECTOR_INT(n_a);
VECTOR_INT WSPT_B = DIM_VECTOR_INT(n_a);
sort_vector(v_a, WSPT_a, n_a, 'D');
sort_vector(v_b, WSPT_B, n_a, 'D');

VECTOR_INT WSPT_b = DIM_VECTOR_INT(n_a);
for(j=0; j<n_a; j++)
{
   WSPT_b[j]=WSPT_B[j]+n_a;
}

for(j=0;j<n_a;j++){
   WSPT[j]=WSPT_b[j];
}
```

```
j=n_a;
for(k=0; k<n_a; k++)
{
   if(j<n)
   {
      WSPT[j] = WSPT_a[k];
      j++;
   }
}

//COMPLETION TIME
VECTOR_INT CT = DIM_VECTOR_INT(n);
CT = completiontime(n,WSPT,tiempos_proceso);

//METAHEURISTICA VNS

int n_iter = 0; //Numero de iteraciones total
float delta = 0;
double FO_A_WSPT = FObjConjuntoA (n, n_a, w, WSPT, CT);
int mejora;
double mejor_1 = FO_A_WSPT;
VECTOR_INT x1 = DIM_VECTOR_INT(n);
VECTOR_INT vecino_mejor = DIM_VECTOR_INT(n);
copy_vector(WSPT,vecino_mejor,n);

while(delta<=0.5)
{
   eps = eps*(1-delta);
   mejor_1 = FO_A_WSPT;
   copy_vector(WSPT,vecino_mejor,n);
   n_iter=0;
   while(n_iter<100000)
   {
      mejora=1;
      while(mejora==1)
      {
         mejora=0;
```

```
    x1 = v1_swap(n, vecino_mejor);
    CT = completiontime(n,x1,tiempos_proceso);
    double FO_A1 = FObjConjuntoA (n, n_a, w, x1, CT);
    double FO_B1 = FObjConjuntoB (n, n_a, w, x1, CT);
    if(FO_B1<=eps)
    {
      if(FO_A1<=mejor_1)
      {
        mejor_1 = FO_A1;
        copy_vector(x1,vecino_mejor,n);
        mejora=1;
      }
    }
}

VECTOR_INT x2 = DIM_VECTOR_INT(n);
mejora=1;
while(mejora==1)
{
    mejora=0;
    x2 = v2_insert(n, vecino_mejor);
    CT = completiontime(n,x2,tiempos_proceso);
    double FO_A2 = FObjConjuntoA (n, n_a, w, x2, CT);
    double FO_B2 = FObjConjuntoB (n, n_a, w, x2, CT);
    if(FO_B2<=eps)
    {
      if(FO_A2<=mejor_1)
      {
        mejor_1 = FO_A2;
        copy_vector(x2,vecino_mejor,n);
        mejora=1;
        while(mejora==1)
        {
          mejora=0;
          x2 = v1_swap(n, vecino_mejor);
          CT = completiontime(n,x2,tiempos_proceso);
          double FO_A2 = FObjConjuntoA (n, n_a, w, x2, CT);
```

```
                double FO_B2 = FObjConjuntoB (n, n_a, w, x2, CT);
                if(FO_B2<=eps)
                {
                   if(FO_A2<=mejor_1)
                   {
                      mejor_1 = FO_A2;
                      copy_vector(x2,vecino_mejor,n);
                      mejora=1;
                   }
                }
             }
             mejora=1;
          }
       }
    }

    VECTOR_INT x3 = DIM_VECTOR_INT(n);
    mejora=1;
    while(mejora==1)
    {
       mejora=0;
       VECTOR_INT q = DIM_VECTOR_INT(n);
       copy_vector(vecino_mejor,q,n);
       x3 = v3_learner(n, q);
       CT = completiontime(n,x3,tiempos_proceso);
       double FO_A3 = FObjConjuntoA (n, n_a, w, x3, CT);
       double FO_B3 = FObjConjuntoB (n, n_a, w, x3, CT);
       if(FO_B3<=eps)
       {
          if(FO_A3<=mejor_1)
          {
             mejor_1 = FO_A3;
             copy_vector(x3,vecino_mejor,n);
             mejora=1;
             while(mejora==1)
             {
                mejora=0;
```

```
        x3 = v1_swap(n, vecino_mejor);
        CT = completiontime(n,x3,tiempos_proceso);
        double FO_A3 = FObjConjuntoA (n, n_a, w, x3, CT);
        double FO_B3 = FObjConjuntoB (n, n_a, w, x3, CT);
        if(FO_B3<=eps)
        {
            if(FO_A3<=mejor_1)
            {
                mejor_1 = FO_A3;
                copy_vector(x3,vecino_mejor,n);
                mejora=1;
            }
        }
    }
    mejora=1;
    while(mejora==1)
    {
        mejora=0;
        x3 = v2_insert(n, vecino_mejor);
        CT = completiontime(n,x3,tiempos_proceso);
        double FO_A3 = FObjConjuntoA (n, n_a, w, x3, CT);
        double FO_B3 = FObjConjuntoB (n, n_a, w, x3, CT);
        if(FO_B3<=eps)
        {
            if(FO_A3<=mejor_1)
            {
                mejor_1 = FO_A3;
                copy_vector(x3,vecino_mejor,n);
                mejora=1;
                while(mejora==1)
                {
                    mejora=0;
                    x3 = v1_swap(n, vecino_mejor);
                    CT = completiontime(n,x3,tiempos_proceso);
                    double FO_A3 = FObjConjuntoA (n, n_a, w, x3, CT);
                    double FO_B3 = FObjConjuntoB (n, n_a, w, x3, CT);
                    if(FO_B3<=eps)
```

```
                        {
                          if(FO_A3<=mejor_1)
                          {
                            mejor_1 = FO_A3;
                            copy_vector(x3,vecino_mejor,n);
                            mejora=1;
                          }
                        }
                      }
                    }
                  }
                }

                mejora=1;
              }
            }
          }

          n_iter++;

        }
        delta = delta + 0.1;

        output(argv[1],argv[2],n,vecino_mejor, mejor_1, eps);

    }
    return 0;
}

int *processtime(int N, int M, MAT_INT PT)
{
    int j,i;
    int *resultado = malloc(N*sizeof(int));

    for(j=0; j<N; j++)
    {
        resultado[j]=0;
```

```
        for(i=0; i<M; i++)
        {
            resultado[j]+=PT[i][j];
        }
    }
    return resultado;
    free(resultado);
    resultado=NULL;
    free_matrix(PT,M);
}

double FObjConjuntoA (int N, int M, VECTOR_DOUBLE Wj, VECTOR_INT orden, VECTOR_INT
Ctime)
{
    int i, j, k;
    double W[N];
    for(j=0; j<N; j++)
    {
        W[j]=Wj[orden[j]];
    }
    //FUNCION OBJETIVO CONJUNTO A

    VECTOR_DOUBLE WA = DIM_VECTOR_DOUBLE(M);
    VECTOR_DOUBLE WB = DIM_VECTOR_DOUBLE(M);
    i=0;
    k=0;
    int CA[M];
    int CB[M];
    for(j=0; j<N; j++)
    {
        if(orden[j]<M)
        {
            CA[i]=Ctime[j];
            WA[i]=W[j];
            i++;
        }
        else
```

```
      {
        CB[k]=Ctime[j];
        WB[k]=W[j];
        k++;
      }
    }
    double FOA=0;
    for(j=0; j<M; j++)
    {
      FOA=FOA+WA[j]*CA[j];
    }
    return FOA;

}

double FObjConjuntoB (int N, int M, VECTOR_DOUBLE Wj, VECTOR_INT orden, VECTOR_INT Ctime)
{
    int j,i,k;
    double W[N];
    for(j=0; j<N; j++)
    {
      W[j]=Wj[orden[j]];
    }

    //FUNCION OBJETIVO CONJUNTO B

    VECTOR_DOUBLE WA = DIM_VECTOR_DOUBLE(M);
    VECTOR_DOUBLE WB = DIM_VECTOR_DOUBLE(M);
    i=0;
    k=0;
    int CA[M];
    int CB[M];
    for(j=0; j<N; j++)
    {
      if(orden[j]<M)
      {
        CA[i]=Ctime[j];
```

```
            WA[i]=W[j];
            i++;
        }
        else
        {
            CB[k]=Ctime[j];
            WB[k]=W[j];
            k++;
        }
    }

    double FOB=0;
    for(j=0; j<M; j++)
    {
        FOB=FOB+WB[j]*CB[j];
    }
    return FOB;

}

int *v1_swap(int N, VECTOR_INT A)
{
    int *resultado = malloc(N*sizeof(int));
    copy_vector(A,resultado,N);
    int a1 = rand()%N;
    int a2 = rand()%N;
    if(a1==a2)
    {
        a1 = rand()%N;
        a2 = rand()%N;
    }
    resultado[a1]=A[a2];
    resultado[a2]=A[a1];
    return resultado;
    free(resultado);
    resultado=NULL;
}
```

```
int *v2_insert(int N, VECTOR_INT B)
{
   int *resultado = malloc(N*sizeof(int));
   copy_vector(B,resultado,N);
   int random1 = rand()%N;
   int random2 = rand()%N;
   int a = search_vector(resultado,N,random1);
   extract_vector(resultado,N,a);
   insert_vector(resultado,N,B[a],random2);
   return resultado;
   free(resultado);
   resultado=NULL;
}

int *v3_learner(int N, VECTOR_INT C)
{
   int i, j;
   int *resultado = malloc(N*sizeof(int));
   int L[N];
   for(i=0; i<N; i++)
   {
      L[i]=10000;
   }

   VECTOR_INT sec = DIM_VECTOR_INT(N);
   VECTOR_INT copia_sec = DIM_VECTOR_INT(N);
   randSequence(sec,N);
   copy_vector(sec,copia_sec,N);
   int aleat = rand()% N/2 + 1;
   VECTOR_INT aleatorio = DIM_VECTOR_INT(aleat);
   for(i=0; i<aleat; i++)
   {
      int num=rand()%N;
      if(i>0)
      {
         for(j=0; j<i; j++)
```

```
                {
                    if(num==aleatorio[j])
                    {
                        num=rand()%N;
                        j=-1;
                    }
                }
            }
            aleatorio[i]=num;
        }
        int c1, c2;
        for(i=0; i<aleat; i++)
        {
            c1 = search_vector(sec,N,aleatorio[i]);
            c2 = search_vector(C,N,aleatorio[i]);
            extract_vector(C,N,c2);
            extract_vector(L,N,c1);
            extract_vector(copia_sec,N,c1);
            insert_vector(L,N,aleatorio[i],c1);
        }
        j=0;
        for(i=0; i<N; i++)
        {
            if(sec[i]!=L[i])
            {
                extract_vector(L,N,i);
                insert_vector(L,N,C[j],i);
                j++;
            }
        }
        copy_vector(L,resultado,N);
        return resultado;
        free(resultado);
        resultado=NULL;
    }

void output(char*entrada,char*salida,int N,VECTOR_INT secuenciafinal,double FO_final, double epsilon)
```

```
{

   FILE *fichero;
   int j;
   fichero = fopen(salida, "a");
   if(fichero==NULL)
   {
      printf("\nNo se puede abrir el archivo de salida\n");
      return -1;
   }
   else
   {
      fprintf(fichero,"\nSecuencia= ");
      for(j=0; j<N; j++)
      {
         fprintf(fichero,"%d ",secuenciafinal[j]);
         fprintf(fichero,"\nValor %d de la secuencia final:: %d \n", j, secuenciafinal[j]);
      }
      fprintf(fichero, "\nEpsilon vale: %f\n", epsilon);
      fprintf(fichero,"%f\n", FO_final);
   }
   fclose(fichero);

}

int *completiontime(int N, VECTOR_INT orden, MAT_INT PT)
{
   int j,i;
   int *resultado = malloc(N*sizeof(int));
   VECTOR_INT C1 = DIM_VECTOR_INT(N);
   VECTOR_INT C2 = DIM_VECTOR_INT(N);
   VECTOR_INT p1 = DIM_VECTOR_INT(N);
   VECTOR_INT p2 = DIM_VECTOR_INT(N);

   for(j=0; j<N; j++)
   {
      i=0;
```

```
        p1[j]=PT[i][orden[j]];
        i=1;
        p2[j]=PT[i][orden[j]];

    }

    C1[0]=p1[0];
    C2[0]=C1[0]+p2[0];

    for(j=1; j<N; j++)
    {
        C1[j]=C1[j-1]+p1[j];
    }
    for(j=1; j<N; j++)
    {
        if(C2[j-1]<=C1[j])
        {
            C2[j]=C1[j]+p2[j];
        }
        else
        {
            C2[j]=C2[j-1]+p2[j];
        }
    }

    copy_vector(C2,resultado,N);
    return resultado;
    free(resultado);
    resultado=NULL;
}


MAT_INT LecturaInstancias(char *filename, int *jobs, int *machs, double *epsilon, int print_flag)
{
    FILE *data;
    int temp_data;
    register int i,j;
```

```
MAT_INT pt;
VECTOR_DOUBLE W;

if (!(data = fopen(filename,"rt")))
   error(2,STOP, filename);

fscanf(data,"%d\n", &temp_data);
*(machs) = temp_data;
fscanf(data,"%d\n", &temp_data);
*(jobs) = temp_data;

pt = DIM_MAT_INT((*machs),(*jobs));
W = DIM_VECTOR_DOUBLE((*jobs));

for(i=0; i<(*machs); i++)
{
   for(j=0; j<*(jobs); j++)
   {
      fscanf(data,"%10d", &temp_data );
      pt[i][j] = temp_data;
   }
   fscanf(data,"\n");
}

for(j=0; j<(*jobs); j++){
   fscanf(data,"%10d", &temp_data );
   W[j]=temp_data;
}
fscanf(data,"\n");

fscanf(data,"%d\n", &temp_data);
*(epsilon) = temp_data;
fclose(data);

if(print_flag == YES)
{
   printf("Instance %s - n: %d    m: %d\n",filename, *(jobs), *(machs));
```

```
        print_int_matrix(pt, *(machs), *(jobs) );
        print_double_vector(W,*(jobs));
        printf("EPSILON = %f\n", *(epsilon));
    }
    return pt;

}

VECTOR_DOUBLE LecturaInstancias2(char *filename, int *jobs, int *machs, double *epsilon, int
print_flag)
{
    FILE *data;
    int temp_data;
    register int i,j;
    MAT_INT pt;
    VECTOR_DOUBLE W;

    if (!(data = fopen(filename,"rt")))
        error(2,STOP, filename);

    fscanf(data,"%d\n", &temp_data);
    *(machs) = temp_data;
    fscanf(data,"%d\n", &temp_data);
    *(jobs) = temp_data;

    pt = DIM_MAT_INT((*machs),(*jobs));
    W = DIM_VECTOR_DOUBLE((*jobs));

    for(i=0; i<(*machs); i++)
    {
        for(j=0; j<*(jobs); j++)
        {
            fscanf(data,"%10d", &temp_data );
            pt[i][j] = temp_data;
        }
        fscanf(data,"\n");
    }
```

```
   for(j=0; j<(*jobs); j++){
      fscanf(data,"%10d", &temp_data );
      W[j]=temp_data;
   }
   fscanf(data,"\n");

   fscanf(data,"%d\n", &temp_data);
   *(epsilon) = temp_data;
   fclose(data);

   if(print_flag == YES)
   {
      printf("Instance %s - n: %d    m: %d\n",filename, *(jobs), *(machs));
      print_int_matrix(pt, *(machs), *(jobs) );
      print_double_vector(W,*(jobs));
      printf("EPSILON = %f\n", *(epsilon));
   }
   return W;
}
```

7.2 Código de generación de instancias

```
#include <schedule.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

double FObjConjuntoB (int N, int M, VECTOR_INT Wj, VECTOR_INT orden, VECTOR_INT Ctime);
int *completiontime(int N, VECTOR_INT orden, MAT_INT PT);

int main(int argc, char *argv[])
{
   int j, i, k;
   int n;
   int m=2;
```

```
n=atoi(argv[1]);
char *salida = argv[2];
int n_a = n/2;
srand(time(NULL));

MAT_INT tiempos_proceso = DIM_MAT_INT(m,n);
VECTOR_INT w = DIM_VECTOR_INT(n);

for(j=0; j<n; j++)
{
   for(i=0;i<m;i++){
      tiempos_proceso[i][j] = rand()%100+1;
   }

}
for(j=0; j<n; j++)
{
   w[j] = rand()%100+1;



}

//WSPT

VECTOR_DOUBLE suma_p =DIM_VECTOR_DOUBLE(n);
for(j=0; j<n; j++)
{
   suma_p[j]=0;
   for(i=0; i<m; i++)
   {
      suma_p[j]+=tiempos_proceso[i][j];
   }
}

VECTOR_DOUBLE v = DIM_VECTOR_DOUBLE(n);
VECTOR_DOUBLE v_a = DIM_VECTOR_DOUBLE(n_a);
VECTOR_DOUBLE v_b = DIM_VECTOR_DOUBLE(n_a);
```

```
for(j=0; j<n; j++)
{
   v[j]=w[j]/suma_p[j];
}
print_vector(v,n);
for(j=0; j<n_a; j++)
{
   v_a[j]=v[j];
}
j=n_a;
for(k=0; k<n_a; k++)
{
   v_b[k]=0;
   if(j<n)
   {
      v_b[k] = v[j];
      j++;
   }
}

VECTOR_INT WSPT_a = DIM_VECTOR_INT(n_a);
VECTOR_INT WSPT_B = DIM_VECTOR_INT(n_a);
sort_vector(v_a, WSPT_a, n_a, 'D');
sort_vector(v_b, WSPT_B, n_a, 'D');

VECTOR_INT WSPT_b = DIM_VECTOR_INT(n_a);
for(j=0; j<n_a; j++)
{
   WSPT_b[j]=WSPT_B[j]+n_a;
}
print_vector(WSPT_b,n_a);

VECTOR_INT WSPT = DIM_VECTOR_INT(n);
for(j=0;j<n_a;j++){
   WSPT[j]=WSPT_a[j];
}
j=n_a;
```

```
for(k=0; k<n_a; k++)
{
   if(j<n)
   {
      WSPT[j] = WSPT_b[k];
      j++;
   }
}
print_vector(WSPT,n);

VECTOR_INT CT = DIM_VECTOR_INT(n);
CT = completiontime(n, WSPT, tiempos_proceso);
print_vector(CT,n);

VECTOR_INT sec = DIM_VECTOR_INT(n);
randSequence(sec,n);

long int eps;
eps = FObjConjuntoB (n, n_a, w, sec, CT);
printf("Epsilon vale: %ld\n", eps);

FILE *fichero;
fichero = fopen(salida , "a");
if(fichero==NULL){
   printf("ERROR\n");
}
fprintf(fichero,"2\n");
fprintf(fichero,"%d\n", n);
for(j=0;j<n;j++){
   fprintf(fichero,"%d ", tiempos_proceso[0][j]);
}
fprintf(fichero,"\n");
for(j=0;j<n;j++){
   fprintf(fichero,"%d ", tiempos_proceso[1][j]);
}
fprintf(fichero,"\n");
for(j=0;j<n;j++){
```

```
    fprintf(fichero,"%d ", w[j]);
  }
  fprintf(fichero,"\n%ld \n", eps);
  fclose(fichero);
}

int *completiontime(int N, VECTOR_INT orden, MAT_INT PT)
{
  int j,i;
  int *resultado = malloc(N*sizeof(int));
  VECTOR_INT C1 = DIM_VECTOR_INT(N);
  VECTOR_INT C2 = DIM_VECTOR_INT(N);
  VECTOR_INT p1 = DIM_VECTOR_INT(N);
  VECTOR_INT p2 = DIM_VECTOR_INT(N);

  for(j=0; j<N; j++)
  {
    i=0;
    p1[j]=PT[i][orden[j]];
    i=1;
    p2[j]=PT[i][orden[j]];

  }

  C1[0]=p1[0];
  C2[0]=C1[0]+p2[0];

  for(j=1; j<N; j++)
  {
    C1[j]=C1[j-1]+p1[j];
  }
  for(j=1; j<N; j++)
  {
    if(C2[j-1]<=C1[j])
    {
      C2[j]=C1[j]+p2[j];
    }
```

```
        else
        {
            C2[j]=C2[j-1]+p2[j];
        }
    }

    copy_vector(C2,resultado,N);
    print_vector(C2,N);

    return resultado;
    free(resultado);
    resultado=NULL;
}

double FObjConjuntoB (int N, int M, VECTOR_INT Wj, VECTOR_INT orden, VECTOR_INT Ctime)
{
    int j,i,k;
    double W[N];
    for(j=0; j<N; j++)
    {
        W[j]=Wj[orden[j]];
    }

    //FUNCION OBJETIVO CONJUNTO B

    VECTOR_INT WA = DIM_VECTOR_DOUBLE(M);
    VECTOR_INT WB = DIM_VECTOR_DOUBLE(M);
    i=0;
    k=0;
    int CA[M];
    int CB[M];
    for(j=0; j<N; j++)
    {

        if(orden[j]<M)
        {
            CA[i]=Ctime[j];
```

```
        WA[i]=W[j];
        i++;
      }
      else
      {
        CB[k]=Ctime[j];
        WB[k]=W[j];
        k++;
      }
    }

    long int FOB=0;
    for(j=0; j<M; j++)
    {
      FOB=FOB+WB[j]*CB[j];
    }
    return FOB;


}
```