Trabajo Fin de Grado
Grado en Ingeniería de Tecnologías Industriales

Análisis de metaheurística híbrida HPV aplicada en entorno de tipo Jobshop con objetivos sostenibles

Autor: Francisco Ojeda Carmona
Tutor: Paz Pérez González

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

Sevilla, 2024

Trabajo Fin de Grado
Grado en Ingeniería de Tecnologías Industriales

Análisis de metaheurística híbrida HPV aplicada en entorno de tipo Jobshop con objetivos sostenibles

Autor:
Francisco Ojeda Carmona

Tutor:
Paz Pérez González

Dpto. de Organización Industrial y Gestión de Empresas I
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2024

Trabajo Fin de Grado: Análisis de metaheurística híbrida HPV aplicada en entorno de tipo Jobshop con objetivos sostenibles

Autor: Francisco Ojeda Carmona

Tutor: 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, 2024

El Secretario del Tribunal

Agradecimientos

A Paz Pérez, tutora de este proyecto por la confianza y la ayuda en la realización de este trabajo.

A mis familiares por estar siempre a mi lado.

Francisco Ojeda Carmona

Sevilla, 2024

Resumen

Una de las principales metas fijadas por la programación de la producción es el de conseguir alcanzar ciertos niveles de eficiencia energética. En lo que concierne a la fabricación, cada vez adquiere más relevancia el buen uso de las fuentes de energía, ya que, el malgasto energético, puede llevar a las empresas a no cumplir con las normas básicas de sostenibilidad y desarrollo medioambiental y tener unos costes energéticos que les impida competir en el mercado.

En términos de estudio, esta consideración energética puede reflejarse dentro de un problema de optimización o bien minimizando el tiempo máximo de terminación de los trabajos (*makespan*), o bien minimizando la suma de tiempos ociosos entre los trabajos para todas las máquinas (*Core Idle Time*).

En este trabajo, se plantea un problema de entorno tipo taller (*Jobshop*) en el cual se fijan como funciones objetivo las dos casuísticas mencionadas anteriormente (minimización de makespan y minimización de Core Idle Time). Para la resolución, se opta por utilizar y comparar las siguientes tres metaheurísticas: AGS (*Algoritmo Genético Simple*), SA (*Simulated Annealing*) y el algoritmo híbrido HPV (*Hybrid PSO -VNS*), que surge de la combinación de las metaheurísticas PSO (*Particle Swarm Operation*) y VNS (*Variable Neighbourhood Search*).

Para la realización del análisis, se ha codificado el problema en lenguaje Python y se han estudiado una batería de instancias Jobshop conocidas en el mundo de la programación como son las instancias de Fisher y Thomson (FT) y Lawrence (LA).

Una vez se tienen los resultados numéricos en Microsoft Excel de los tres algoritmos para cada función objetivo, se demuestra que el algoritmo de estudio HPV presenta mejoras frente a los algoritmos AGS y SA y que, la función objetivo de minimización de tiempos ociosos entre los trabajos, puede conducir a una solución distinta de la que proporciona la minimización de makespan. La elección de qué función objetivo resulta más sostenible u económica dependerá, para cada caso particular, del coste que supone pausar las máquinas, qué máquinas se pausan o de la prioridad dada al makespan.

Abstract

One of the main goals set by production programming is to achieve certain levels of energy efficiency. As far as manufacturing is concerned, the proper use of energy sources is becoming more and more important, since energy waste can lead companies to not comply with the basic standards of sustainability and environmental development and has higher costs.

In terms of study, this energy consideration can be reflected within an optimization problem either by minimizing the maximum completion time of each job (*makespan*), or by minimizing the sum of core idle times for all machines (*Core Idle Time).*

In this work, a problem framed in a *Jobshop* layout is proposed in which the two cases mentioned above (makespan minimization and weighted Core Idle Time minimization) are set as objective functions. To solve the problem, we use three metaheuristics: AGS (Simple Genetic Algorithm), SA (Simulated Annealing) and HPV (Hybrid *PSO-VNS*), which is born from the combination of the PSO (Particle Swarm Operation) and VNS (Variable Neighborhood Search) metaheuristics.

To carry out the analysis, the problem has been coded in Python language and a well-known set of instances for the Jobshop problem in the programming world have been studied, such as the Fisher and Thomson (FT) and Lawrence (LA) instances.

Once the numerical results are obtained in Microsoft Excel for the three algorithms for each objective function, it is shown that the HPV study algorithm presents improvements over the AGS and SA algorithms. On the other hand, the objective function for minimizing core idle times can lead to a different solution than the one provided by the makespan minimization. The choice of which objective function is more sustainable or economical will depend, for each case, on the cost of determine the time that machines are in stand by, which machines are idle, or the priority given to the makespan.

Tabla de contenido Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

La gestión de recursos cada vez juega un papel más importante dentro de las industrias. Hoy en día, la fuerte competencia dentro del sector industrial, junto con la importancia del desarrollo sostenible, motivan el estudio de la búsqueda de eficiencia y optimización en los procesos de fabricación. Dentro de esta búsqueda, este proyecto se centra en la programación de operaciones como vía de mejora de dichos procesos.

La programación de operaciones se define como la asignación temporal de recursos para la fabricación de un conjunto de productos. Como resultado, se obtiene un programa de producción (production schedule) que Contiene información sobre cuándo debe comenzar cada recurso a procesar qué producto (Pérez González, Fernández-Viagas & Framiñán, 2021).

Dicha asignación de recursos se lleva a cabo en base a múltiples factores, dependiendo de las características del problema que se esté considerando. Así, el número de maquinaria disponible, cantidad de productos a procesar, recursos humanos, costes de producción, objetivos y estrategias entre otros más factores, conducen a un problema matemático particularizado con un amplio abanico de soluciones posibles.

En la práctica, de la gran cantidad de restricciones que pueda contener un problema, unido a la multitud de soluciones que se puedan aportar, surge la necesidad de no afrontar el problema manualmente sino a través de una estructura matemática que posteriormente pueda ser interpretada por un software o lenguaje de programación.

En este trabajo, el objetivo es representar un problema de tipo taller (Jobshop) donde se dispone de una determinada cantidad de máquinas que han de procesar una determinada cantidad de trabajos con diferentes rutas de proceso. La restricción que supone que cada trabajo tenga una ruta diferente, hace que el problema, en términos matemáticos, adquiera mayor complejidad y esté considerado como uno de los problemas más difíciles de resolver dentro del ámbito de la programación de operaciones.

Para la resolución del problema, existen varias técnicas extendidas destacando la programación lineal y el uso de algoritmos aproximados o metaheurísticas. El núcleo central de este trabajo consiste en el desarrollo y comparación de varias metaheurísticas avanzadas estableciendo un riguroso análisis y determinando, de los algoritmos seleccionados, cual aporta mejores soluciones para un conjunto de situaciones distintas (instancias).

Referente a la estructura del presente documento, consta de cinco capítulos y un anexo con la codificación en Python:

- Capítulo 1: Introducción. Presente apartado, se enuncia el motivo de estudio junto con los objetivos que se pretenden conseguir en el proyecto.
- Capítulo 2: Programación de operaciones. Contenido teórico para introducir al lector los conceptos que componen la base de la programación. El último apartado de este capítulo pretende clasificar el problema a tratar una vez se conoce la base teórica.
- Capítulo 3: Problema considerado. En este capítulo se describe el tipo de problema y algunos de los distintos métodos aproximados para la resolución. Se profundizará en la metaheurística híbrida HPV y en dos algoritmos clásicos como el Algoritmo Genético (AG) y Simulated Annealing (SA)
- Capítulo 4: Implementación de los métodos. Se define como se han codificado los programas en Python y se realiza un análisis de los datos obtenidos.
- Capítulo 5: Conclusiones. Ideas finales que se desgranan del análisis de resultados y validez de la investigación.
- Bibliografía. Referencias a artículos, libros y páginas de internet.
- Anexo I: Códigos realizados en lenguaje Python

2 PROGRAMACIÓN DE OPERACIONES

Para adentrarse dentro del marco teórico de la programación de operaciones o *scheduling*, es preciso conocer ciertos conceptos que son la base para entender cualquier problema de programación. En este segundo capítulo, junto con la definición de estos conceptos básicos, se muestra cómo se modela un problema y algunos de los métodos utilizados para su resolución.

2.1 Conceptos básicos

A como se muestra en el libro de apuntes de la asignatura Programación de Operaciones de la Escuela Técnica Superior de Ingeniería de Sevilla (Pérez González, Fernández-Viagas & Framiñán, 2021), en cualquier problema de programación se pueden identificar estos términos:

- Maquinaria disponible (*Machine*). Recurso productivo con capacidad para realizar operaciones de transformación/transporte de material. En la adaptación a la práctica, puede representar cualquier tipo de maquinaria (horno/fresadora/martillo/máquina de corte/camión/elevadora/carretilla…), trabajadores (un operario o conjunto de operarios) o ambos (una planta donde los operarios trabajan con determinadas máquinas).
- Trabajos (*Jobs*). Producto que es objeto de una operación en alguna de las máquinas de la fábrica. Puede representar distintos objetos físicos como, por ejemplo, un tornillo, una botella, componentes de un avión o un lote de productos entre otros.
- Tiempo de proceso (*Processing time*). Duración temporal de operación de un trabajo en una máquina.
- Secuencia (*Sequence*). Orden de procesamiento de todos los trabajos en las máquinas. En el modelo de programación, se representa mediante un vector cuyas componentes varían en función de las características del problema, como se muestra más adelante. En un ejemplo en el que se tenga una sola máquina procesando cinco trabajos distintos, la secuencia solución puede ser (2,3,1,5,4). En esta secuencia de ejemplo el trabajo 2 sería el primero seguido del trabajo 3, posteriormente 1 … y finalizando el procesado con el trabajo 4.
- Ruta de proceso (*Route*). Orden en el que cada trabajo es procesado en las máquinas de las que se dispone. Se representa con un vector de tamaño igual al número de máquinas. Así, por ejemplo, si se dispone de tres máquinas distintas (máquinas 1,2 y 3) y se tiene un trabajo A que tiene que ser procesado primero en la máquina 1, posteriormente en 3 y para finalizar en 2, la ruta de proceso del trabajo A es (1,3,2).
- Programa (*Schedule*). Asignación en la escala temporal concreta de las máquinas de una empresa para procesar un conjunto de trabajos.

  En general, un programa determina el comienzo y el fin de una operación a realizar en cada recurso productivo. En muchos casos, puesto que el tiempo de proceso es conocido, es suficiente conocer el tiempo de comienzo o el tiempo de fin. No obstante, si existen operaciones interrumpibles, esta información puede no ser suficiente, por tanto, en este caso puede ser preciso establecer los instantes de interrupción y continuación de las operaciones.

  En función de cómo un programa cumple con las restricciones de un modelo, se establecen estos dos tipos:

  a) Admisible (*Feasible schedule*). El programa cumple con todas las restricciones del modelo.

  b) Admisible Semiactivo (*Semi-active schedule*). El programa cumple con todas las restricciones y, además, representa la solución donde los trabajos se procesan lo antes posible o, en otras palabras, no es posible adelantar un trabajo para un orden de procesamiento fijado.

Para la representación de un programa, se utilizan diagramas de Gantt. Un diagrama de Gantt es un gráfico de barras que consta de dos ejes, uno vertical, en el cual se representa el número de recursos/máquinas y un eje horizontal que representa unidades temporales (segundos, minutos, días, etc.).

El objetivo que se consigue utilizando esta herramienta, es que se puede observar el orden en que cada trabajo es procesado en cada máquina y los tiempos de inicio y finalización de cada uno de ellos, es decir, se puede ilustrar visualmente la solución que adopta el programa.

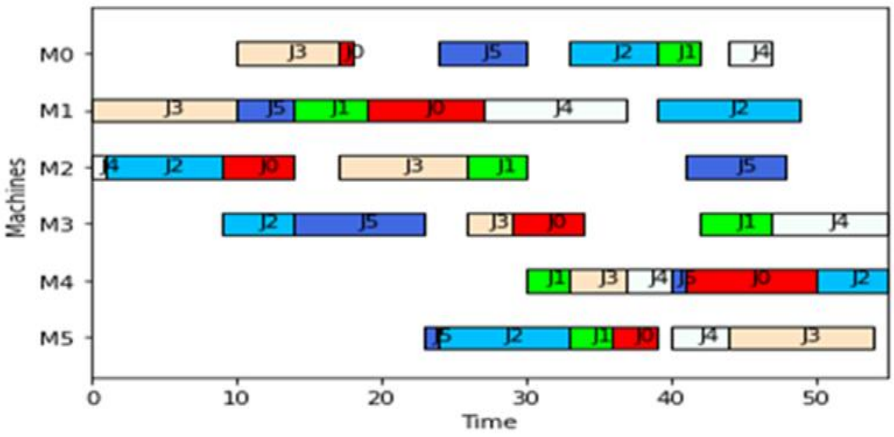


Figura 1. Ejemplo de diagrama de Gantt. [Fuente: Elaboración propia].

En el ejemplo de la Figura 1, se puede apreciar un problema de un taller compuesto por seis máquinas (M0, M1, M2, M3, M4, M5) y tres trabajos a procesar (J0, J1, J2, J3, J4, J5). En el diagrama aparecen las máquinas en el eje vertical mientras que el eje horizontal representa unidad de tiempo. De esta forma, se conoce cuando se inicia y finaliza el procesado de todos los trabajos en todas las máquinas.

2.2 Modelos de programación

Un modelo de programación es la abstracción formal de un problema de programación de la producción cuya solución es un programa de producción (Pérez González, Fernández-Viagas & Framiñán, 2021). Para analizar los modelos, en este documento se muestra la notación más extendida y estandarizada conocida como "Notación de Graham" (Graham et al, 1979). La notación de Graham estructura los modelos en tres factores diferenciados α | β | γ: Entornos (α), restricciones (β) y objetivos (γ).

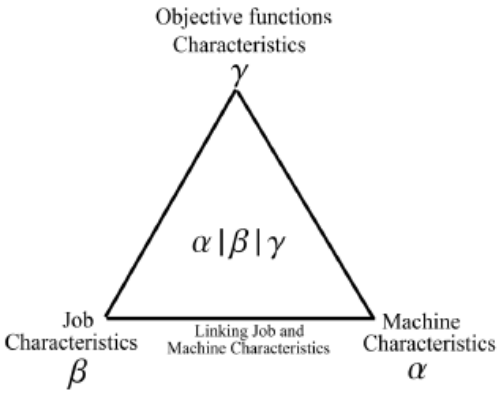


Figura 2. Partes que componen un modelo de programación. [Fuente: "Manufacturing Scheduling Systems: An interated view on Models, Methods and Tools" (Framiñán JM, Leisten R y Ruiz R , 2014)]

A continuación, en los siguientes subcapítulos se detallará en profundidad cada uno de los citados factores que componen el modelo.

2.2.1 Entornos (α)

Los entornos hacen referencia a los recursos que dispone una fábrica. Así, se diferencia un entorno en base al número de recursos que posea una empresa y a las características de estos recursos.

Para una explicación más didáctica, en los ejemplos que se muestran a continuación se hace la suposición de máquinas como recursos de fabricación. Los tipos de entornos más conocidos son los siguientes:

- Una máquina (*Single Machine*) α=1 | β | γ. En este caso, la fábrica dispone de una sola máquina la cual va procesando trabajos. Estos trabajos, solo contemplan una etapa de procesado en la máquina y en cuanto a cantidad de soluciones posibles (Admisibles semiactivas), este entorno dispone de (n!).

  Una solución, para un entorno de este tipo, viene dada por una secuencia simple, que corresponde a un vector con tantas componentes como número de trabajos a procesar contenga la máquina. De esta forma, a modo de ejemplo, una máquina que ha de procesar 2 trabajos diferentes (J1 y J3) tendría (2!) Soluciones semiactivas posibles, es decir, dos soluciones posibles (J1, J3) y (J3, J1). Para el caso de adoptar la solución (J3, J1) significaría que el trabajo J3 se realizaría primero seguido del trabajo J1. El diagrama de Gantt de este ejemplo de Single Machine puede verse en la Figura 3.

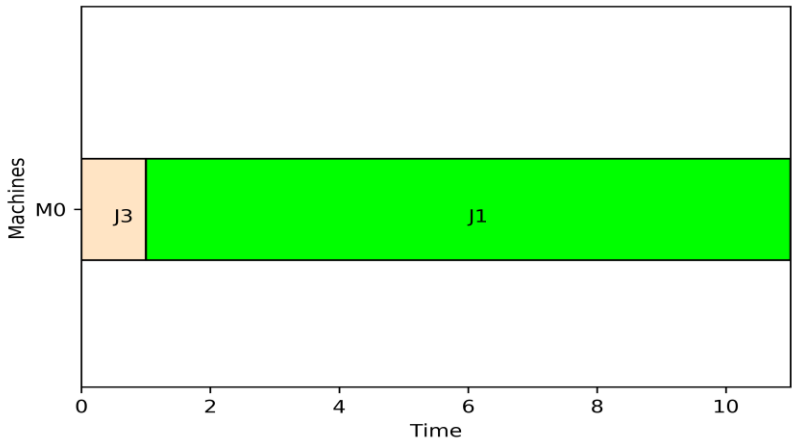


Figura 3. Diagrama de Gantt para un ejemplo de Single Machine. [Fuente: Guía web librería de Python Scheptk (Framiñán, 2022)].

- Máquinas paralelas (*Parallel Machines*) α=Pm/Qm/Rm | β | γ. En este entorno, se trabaja con más de una máquina a la vez con lo que se puede dividir el procesado entre estas máquinas. Según la velocidad que las máquinas procesen los trabajos existen tres tipos:

  a) Máquinas paralelas idénticas (*Identical Parallel Machines*) α=Pm | β | γ. Las máquinas de las que se dispone son idénticas por lo que el tiempo de procesado de cada trabajo en cada una de ellas es el mismo.

  b) Máquinas paralelas relacionadas (*Related Parallel Machines*) α=Qm | β | γ. Entre las máquinas existe una relación en cuanto a tiempo de procesado. Un ejemplo de este caso pudiera ser una fábrica que dispusiera de dos máquinas y una de ellas tardase el doble en procesar cualquier trabajo con respecto a la otra máquina.

  c) Máquinas paralelas no relacionadas (*Unrelated Parallel Machines*) α=Rm | β | γ. Entre las máquinas

no existe relación alguna entre los tiempos de procesado.

En cuanto a codificación de solución, se representa con una secuencia simple, al igual que en un entorno Single Machine, con la única diferencia que habría que fijar una regla de asignación de trabajos, siendo la más común la regla ECT (*Earliest Completion Time*). Con esta regla, los trabajos se irían procesando en las máquinas donde su procesado terminase antes. De esta forma, con una secuencia más una regla de asignación se tiene (n!) soluciones posibles.

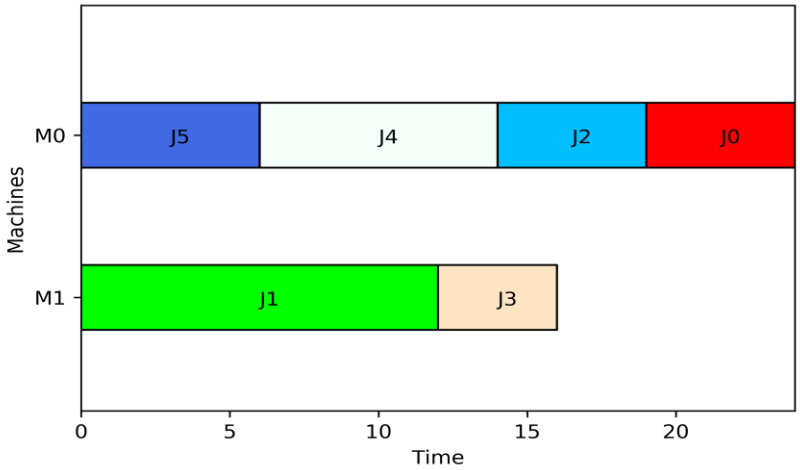


Figura 4. Diagrama de Gantt para un ejemplo de Parallel Machine. [Fuente: Guía web librería de Python scheptk (Framinan, 2022)].

- Entornos de tipo taller. α=Fm/Jm/Om | β | γ. Un tipo taller dispone de un conjunto de máquinas que realizan operaciones distintas. En el caso de los talleres, un trabajo ha de pasar por todas las máquinas y existe una clasificación en función de las rutas de procesado de cada trabajo:

  a) Taller de flujo (*Flowshop*) α=Fm | β | γ. La ruta de todos los trabajos es la misma. Un ejemplo para este caso sería un taller que tuviese dos máquinas A(corte) y B(fresado) y todos los trabajos a procesar en estas máquinas siguiesen el orden (A, B), es decir, para todos los trabajos primero sería la etapa de corte y posteriormente la etapa de fresado.

     Existe un caso particular dentro del taller de flujo denominado Taller de flujo de permutación (*Permutation Flowshop*) α=Fm | β=prmu | γ, que a diferencia del caso estándar añade la restricción de adoptar la misma secuencia en todas las máquinas durante el proceso. Un ejemplo de este entorno se puede observar en la Figura 5, donde la ruta de procesado para todos los trabajos es (M0, M1, M2, M3) y en todas las máquinas se tiene la misma secuencia (J0, J1, J4, J3, J2).

     En cuanto a soluciones, un problema tipo flowshop sin restricciones tiene $(n!)^m$ soluciones mientras que añadiendo la restricción de permutación, el problema reduce el número de soluciones a (n!).

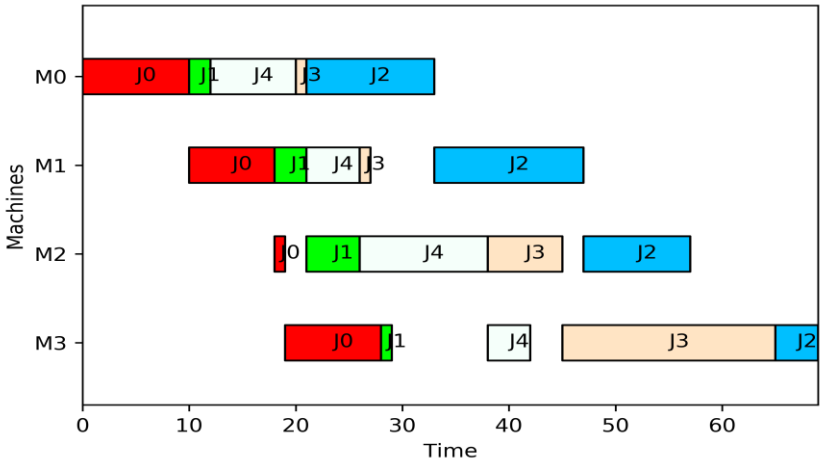


Figura 5. Ejemplo de diagrama de Gantt de un problema de taller de flujo. [Fuente: Guía web librería de Python scheptk (Framinan 2022)].

b) Taller (Jobshop) α=Jm | β | γ. La ruta de procesado para un problema de tipo Jobshop es distinta para cada trabajo. A modo de ejemplo, si se dispone de un taller con dos máquinas A(corte) y B(fresado) los trabajos no van a tener la misma ruta necesariamente por lo que puede ser que un trabajo J1 pase primero por la máquina de corte y posteriormente fresado (A, B) y otro trabajo J2 pase primero por la máquina de fresado y acabe en la máquina de corte (B, A).

   En cuanto a solución, este tipo de problemas tiene como máximo $(n!)^m$. La representación del problema se simplifica utilizando la codificación de secuencia extendida S. La secuencia extendida es un vector compuesto de n x m elementos donde cada trabajo aparece m veces. La aparición en la secuencia extendida de un trabajo significa que ese trabajo se ha de procesar en la máquina que marque el orden de su ruta.

   A modo ilustrativo, supongamos un problema de tres máquinas (M0, M1, M2) y tres trabajos (J0, J1, J2), la ruta de cada trabajo viene dado por R0= (M0, M1, M2), R1= (M2, M0, M1) y R2= (M1, M2, M0). Una solución para este problema expresado como secuencia extendida toma la forma de (2, 1, 0, 2, 1, 0, 1, 2, 0). Analizando la Figura 6 se puede observar cómo se comienza a procesar el trabajo 2 donde marca su ruta de procesado, es decir, en la máquina 1, posteriormente se procesa el trabajo 1 donde marca la ruta del trabajo 1, en la máquina 2 y así sucesivamente hasta completar el proceso.

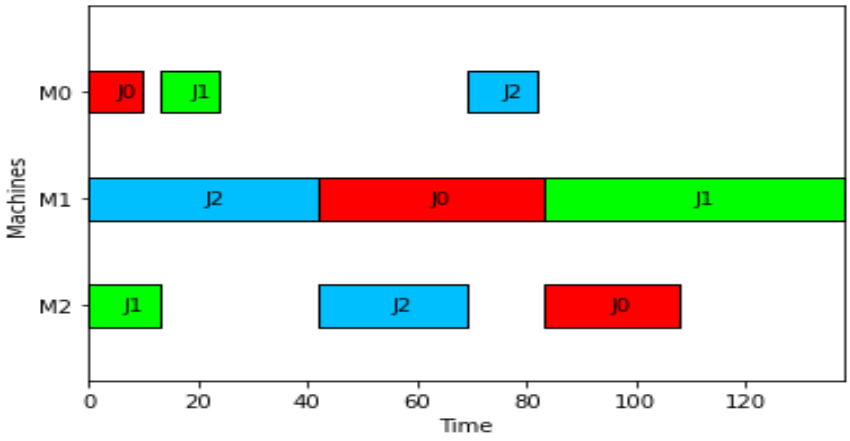


Figura 6. Ejemplo de diagrama de Gantt de un problema Jobshop 3x3. [Fuente: Elaboración propia].

c) Taller abierto (Openshop) α=Om | β | γ . En este caso la ruta de procesado es abierta, es decir no existe una ruta predeterminada. Este caso es el más complejo de analizar. El número de soluciones para un problema de este tipo es de $(m!)(n!)^{m.}$

Para tener una solución es necesario dar una secuencia para cada máquina. A diferencia del Jobshop, no existe ruta por lo que cuando se da el programa se está construyendo la ruta de forma simultánea. La forma de codificación es parecida a la del problema Jobshop, utilizando una secuencia extendida de n x m elementos dada por las operaciones. La visualización en un Gantt es análoga a la de un problema de tipo Jobshop.

2.2.2 Restricciones (β)

Las restricciones de un problema de programación pueden afectar a la solución final estableciendo una o varias características que se deben tener en cuenta dentro del procesado. De base, todos los problemas han de cumplir con las siguientes suposiciones:

- Todos los trabajos están disponibles al principio de la programación
- Los trabajos no se pueden interrumpir
- Las máquinas siempre están disponibles.
- Cada máquina puede hacer un trabajo, y un trabajo puede ser realizado en una máquina
- El buffer entre máquinas se supone infinito
- El tiempo de transporte es despreciable

Por tanto, un problema β = ∅ es aquel que cumple con las citadas suposiciones básicas, pero no tiene ninguna restricción añadida.

Existen muchos tipos de restricciones tomando en consideración determinados factores, algunos de los más comunes son los que se detallan a continuación:

- Tiempos de setup (β = Sijk). Se establece un tiempo de preparación del procesado de cada trabajo en cada máquina previo al propio procesado del trabajo. Los tiempos de setup pueden ser anticipatorios o no anticipatorios dependiendo de si existe la posibilidad de realizar el setup no necesariamente cuando el trabajo se ha de procesar.
- Formación de lote (β = batch). Cuando una máquina puede realizar el procesado de varios trabajos a la vez, estos pueden agruparse por lotes.
- Fechas de entrega (β = dj). Introduce la importancia de entregar cierto trabajo con un plazo de entrega. En el caso de đj, la fecha de entrega de cada trabajo j es de cumplimiento obligatorio.
- Fechas de llegada (β = rj). Muestra la llegada de los trabajos al sistema de procesado. Si el problema presenta esta restricción indica que no todos los trabajos están disponibles al principio.
- Interrupción de operaciones (β = pmtn). La interrupción puede ser debida a indisponibilidad de las máquinas o por mejora de la eficiencia del sistema.
- Precedencias (β = precedence). Un trabajo no puede procesarse hasta que no termine de procesarse otro trabajo existente en el sistema.

Para entornos de tipo taller, existen restricciones adicionales entre las que destacan:

- (β = prmu). Flowshop de permutación. Todas las máquinas del taller tienen la misma secuencia.
- (β = no-idle). No están permitidos tiempos ociosos entre los trabajos en las máquinas.
- (β = no wait). Los trabajos no pueden esperar entre máquinas o etapas de procesado.
- (β = buffer). Cuando existe limitación de almacenaje previo al procesado del trabajo.

2.2.3 Objetivos (γ)

La función objetivo representa la decisión de optimizar o conseguir un determinado objetivo. La solución para una función objetivo en particular se busca que sea máxima o mínima, es decir, se busca minimizar o maximizar un propósito y que alcance un valor fijo o este acotado dentro de un intervalo.

En general, todos los objetivos tienen relación con el tiempo de terminación, aunque existen diferencias en las soluciones obtenidas dependiendo de la importancia que se les concede a ciertos objetivos. Existe una clasificación en cuatro categorías en función de las prioridades: Coste, Tiempo, Calidad y Flexibilidad.

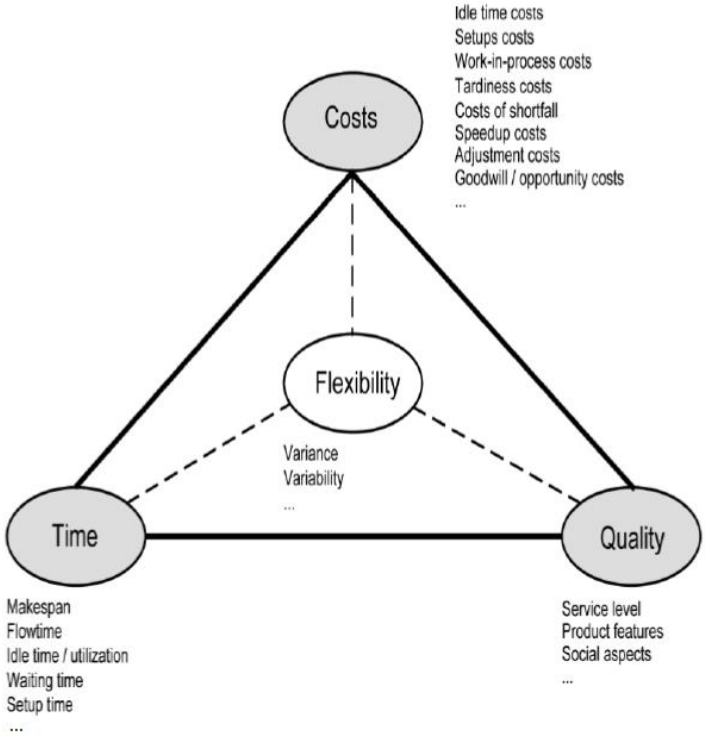


Figura 7. Clasificación con ejemplos de funciones objetivos en base a los 4 aspectos principales. [Fuente: "Manufacturing Scheduling Systems: An interated view on Models, Methods and Tools" (Framiñán JM, Leisten R y Ruiz R , 2014)].

Para establecer funciones objetivo dentro de esta clasificación es indispensable conocer las siguientes medidas básicas:

- C_j (Completion time). Instante de terminación del trabajo j.
- F_j (Flowtime). Tiempo de flujo del trabajo j. Tiempo que permanece el trabajo en el sistema. ($F_j = C_j - r_j$).
- L_j (Lateness). Retraso del trabajo j. ($L_j = C_j - d_j$).
- T_j (tardiness). Tardanza del trabajo j. ($T_j = \max\{0, L_j\}$).
- E_j (earliness). Tiempo de adelanto del trabajo j. ($E_j = \{0, -L_j\}$).
- U_j (Tardy job). Trabajo tardío ($U_j = 1$ si $T_j > 0$ / $U_j = 0$ e.c.c).

Con estas medidas, se pueden establecer funciones objetivo resultado de sumatorios y búsqueda de máximo de estas variables. A modo de ejemplo el makespan representa el valor máximo de C_j ($\max C_j$) o el Total Flowtime es resultado del sumatorio de F_j ($\sum F_j$).

También es posible analizar la importancia de ciertos trabajos o máquinas mediante el uso de ponderaciones. Esto no es más que añadir un elemento W_j, con distinto valor dependiendo de la

importancia del trabajo/máquina, multiplicando en la función objetivo de interés. Así, por ejemplo, $\sum W_j C_j$ representa la suma los tiempos de terminación de los trabajos ponderados, donde cada trabajo adquiere un peso específico que cambia el valor final de la función objetivo.

Desde el punto de vista de la reducción de costes, uno de los principales objetivos a tratar es el de minimizar los tiempos ociosos dentro del procesado. En función de en qué momento del proceso se desee minimizar el tiempo ocioso existen tres tipos:

- Front Idle Time (FIT). La función objetivo fija reducir el tiempo ocioso entre el instante inicial (Tiempo=0) y el momento en que comienza a procesarse el primer elemento. Por tanto, se busca minimizar este tiempo ocioso inicial donde la situación ideal buscada en este caso sería que todos los trabajos comenzaran en el instante (tiempo=0). Su notación sería $\sum FIT_i$ (Sumatorio de FIT para todas las máquinas i), en caso de que cada máquina tuviese distinta ponderación sería $\sum W_i FIT_i$.
- Core Idle Time (CIT). Corresponde al tiempo ocioso existente entre trabajo y trabajo dentro de una misma máquina. Analizando un diagrama de Gantt es el tiempo ocioso intermedio. La función objetivo buscará reducir estos tiempos de "pausa" en las máquinas intentando que el proceso se lleve a cabo de forma continua. Su notación sería $\sum CIT_i$ (Sumatorio de CIT para todas las máquinas i), en caso de que cada máquina tuviese distinta ponderación sería $\sum W_i CIT_i$
- Back Idle Time (BIT). Es el tiempo ocioso que existe entre el tiempo de finalización de un trabajo en su última máquina y el tiempo de finalización de todo el procesado. La función objetivo buscaría reducir estos tiempos y la situación ideal sería aquella donde todos los trabajos terminasen en el mismo instante temporal. Su notación sería $\sum BIT_i$ (Sumatorio de BIT para todas las máquinas i), en caso de que cada máquina tuviese distinta ponderación sería $\sum W_i BIT_i$

En la Figura 8 puede observarse un ejemplo de localización de tiempos ociosos analizando un diagrama de Gantt.

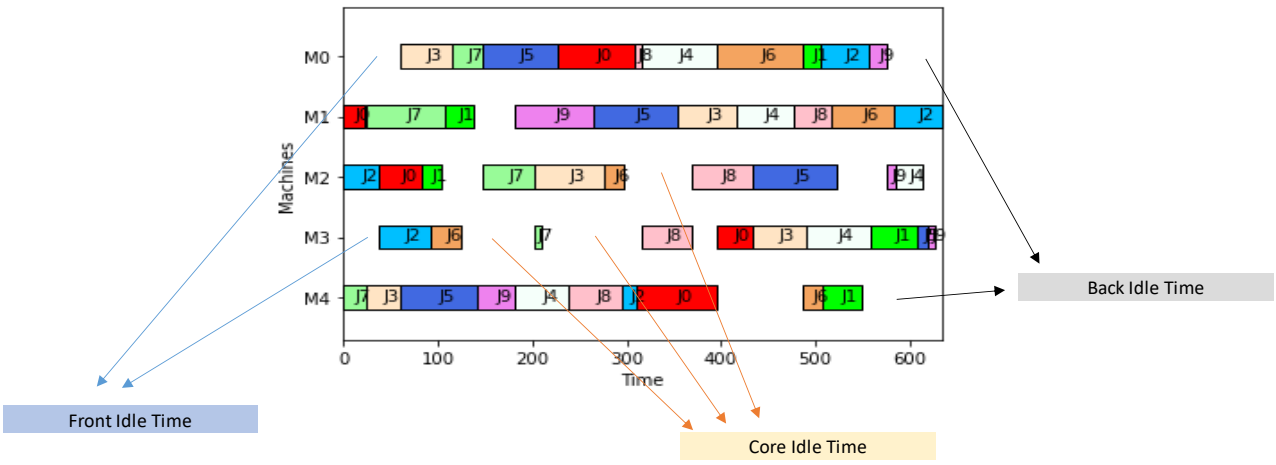


Figura 8. Diagrama de Gantt de un problema Jobshop para ilustrar los tipos de tiempos ociosos. [Fuente: Elaboración propia].

2.3 Métodos de resolución

Actualmente existen diversas técnicas de resolución para los problemas de programación de la producción. La variedad y la dificultad que presentan algunos problemas conduce a utilizar métodos diferentes. En este documento se opta por profundizar en los algoritmos aproximados.

Los algoritmos aproximados, a pesar de que no siempre pueden garantizar la solución óptima, son capaces de proporcionar una buena solución en un intervalo de tiempo razonable para todo tipo de problemas de optimización (B. Melián, J.A. Moreno Pérez, J.M. Moreno Vega, 2003).

En los apuntes de la asignatura Programación de Operaciones (Pérez González, Fernández-Viagas & Framiñán, 2021), se establece la diferenciación de dos tipos de algoritmos dependiendo de su campo de aplicación:

- Heurísticas: Son procedimientos dependientes del problema que se tenga en cuestión. Se suele utilizar para aquellos problemas donde no es posible obtener una solución óptima, con una instancia relativamente sencilla. Un ejemplo de estas son las heurísticas constructivas como Cheapest Insertion, Minimum Slack y NEH.
- Metaheurísticas: A diferencia de las heurísticas, representan un conjunto de técnicas o procedimientos generales aplicables a un amplio abanico de problemas. Las metaheurísticas surgen de la necesidad de abordar el problema con una técnica más avanzada que las heurísticas.

  En función de los procedimientos heurísticos, naturaleza o fuente de inspiración, estrategias de búsqueda etc. existen diversas formas de clasificar tipos de metaheurísticas. En referencia al número de soluciones iniciales y a cómo estas evolucionan siguiendo una trayectoria de búsqueda, es de especial interés la clasificación del artículo *Metaheurísticas: Una visión global* (B. Melián, J.A. Moreno Pérez, J.M. Moreno Vega, 2003). En este artículo se definen dos tipos:

  a) Metaheurísticas basadas en trayectorias. En este caso, se parte de una secuencia solución a la cual se le buscan posibles soluciones mejores dentro de su vecindario. A este tipo de búsqueda de mejores soluciones posibles en secuencias vecinas se le conoce como búsqueda local.

     Para una mejor compresión, se entiende como vecindario el conjunto de soluciones que se desgranan de realizar pequeñas variaciones en la secuencia solución. Por ejemplo, si se tiene como solución a un problema de secuenciación la secuencia (J3, J2, J1, J4), una secuencia vecina surge de realizar algún tipo de cambio como por ejemplo intercambio de dos posiciones. Al realizar el intercambio de las dos primeras posiciones se obtiene la solución vecina (J2, J3, J1, J4). Si el intercambio se lleva a cabo entre más posiciones podemos construir un vecindario de soluciones, así, por ejemplo, la secuencia inicial (J3, J2, J1, J4) puede tener un vecindario compuesto por tres secuencias que surgen del intercambio de la primera posición con las tres restantes: (J2, J3, J1, J4), (J1, J2, J3, J4), (J4, J3, J1, J3). En el caso de que cualquiera de estas secuencias sea mejor que la de inicio, el algoritmo actualizará la nueva mejor solución.

     Algunos de los ejemplos más destacados incluidos dentro de esta clasificación de metaheurísticas son los algoritmos Simulated Annealing(SA), Variable Neighborhood Search (VNS), Tabú Search(TS), Greedy Randomized Adaptive Search Procedure (GRASP) etc.

  b) Metaheurísticas basadas en poblaciones. Las poblaciones componen un conjunto de soluciones de partida, por tanto, a diferencia de las metaheurísticas de búsqueda local, no se parte exclusivamente de una solución a la cual se le realizan cambios, sino que parte de un conjunto de soluciones diversas, estableciendo de esta manera, varios caminos de búsqueda. Como ejemplos destacados se pueden encontrar el Algoritmo Genético (GA), Particle Swarm Optimization (PSO) o Ant colony optimization algorithms (ACO).

3 PROBLEMA CONSIDERADO

En este tercer capítulo, se plantea un modelo para la situación que se pretende analizar y se muestra la relación entre los objetivos a optimizar y la sostenibilidad. En cuanto a métodos de resolución, se hace un breve recorrido de las distintas metaheurísticas utilizadas en el pasado y se profundiza en tres algoritmos seleccionados: Algoritmo Genético (AG), Simulated Annealing (SA) y Hybrid PSO-VNS (HPV). Este último será el que se estudie con más detalle al ser el centro de atención de este proyecto.

3.1 Descripción del problema

3.1.1 Sostenibilidad y objetivos de optimización

En la industria, así como en la sociedad en general, el uso inadecuado de las fuentes de energía adquiere cada vez más conciencia. Esta preocupación, llevó a definir el término desarrollo sostenible por la Comisión Mundial sobre el Medio Ambiente y Desarrollo de las Naciones Unidas en el Informe "Nuestro Futuro Común" (WCED,1987). En este informe se cita textualmente: *"El desarrollo sostenible es el desarrollo que satisface las necesidades de la generación presente sin comprometer la capacidad de las generaciones futuras para satisfacer sus propias necesidades"*. La contaminación, el cambio climático o el agotamiento de algunas fuentes de energía no renovables, son algunas de las consecuencias que pueden empeorar el nivel de vida de futuras generaciones.

Para combatir aquello que conlleva un desarrollo no sostenible, desde la Asamblea General de las Naciones Unidas surgió la *Agenda 2030* (Naciones Unidas, 2015). En ella se promueve, fijando objetivos anuales, el uso de energías renovables, el desarrollo de nuevas tecnologías y un uso eficiente de los recursos energéticos.

La eficiencia energética, por tanto, está estrechamente ligada al término sostenibilidad, ya que su cumplimiento obliga a establecer un consumo responsable y controlado. Una industria que no fuese eficiente estaría dando pie, no solo un incremento en sus costes energéticos, sino a un malgasto que pudiera tener consecuencias graves tanto para el medioambiente como para la economía.

Desde la programación de operaciones también se pueden lograr ciertos objetivos de eficiencia. Por ello, en el caso particularizado de estudio, se establece un modelo en el cual se tienen en cuenta dos objetivos que buscan minimizar el consumo energético de una producción para un problema de tipo Jobshop. Estos objetivos son:

- Minimización del máximo tiempo de terminación de los trabajos (*makespan*). Con este objetivo se busca que la producción termine lo antes posible. Cuanto menor sea la duración del proceso completo, menor tiempo de funcionamiento de las máquinas y por ende mayor ahorro.
- Minimización de la suma de los tiempos ociosos entre dos trabajos en una misma máquina (*Core Idle Time*). Este criterio considera que el tiempo ocioso de una máquina supone un gasto energético. Si las máquinas presentan tiempos de espera entre que se procesan dos trabajos, significa que la máquina se expone a sufrir picos de arranque en la curva de consumo. La situación idónea sería aquella donde una vez las máquinas arrancan no se apaguen hasta que finalizan todas las operaciones que deba realizar. En caso de que pueda existir diferencia entre el gasto de cada una de las máquinas de las que se dispone, el objetivo se puede ponderar fijando un peso para cada máquina (Corte Idle Time Ponderado).

3.1.2 Modelo de problema

En este subapartado se define el modelo de estudio. Para los objetivos sostenibles expuestos, surgen dos problemas que serán enunciados siguiendo la notación de Graham ($\alpha \mid \beta \mid \gamma$).

El entorno sería de tipo Jobshop (J_m), y no se contemplan restricciones añadidas a las suposiciones básicas ($\beta = \emptyset$). En caso de que el entorno fuese de tipo Flowshop, existiría la posibilidad de introducir la restricción β=no-idle, que supondría la obligación de procesar todos los trabajos en las máquinas sin pausas uno detrás de otro. Este caso, para un Jobshop, es imposible que se pueda cumplir siempre ya que el hecho de que cada trabajo tenga una ruta distinta de procesado imposibilita en algunas situaciones que el problema cumpla con dicha restricción. Por este motivo, se busca desde la función objetivo minimizar los tiempos ociosos en lugar de establecer una restricción en el problema.

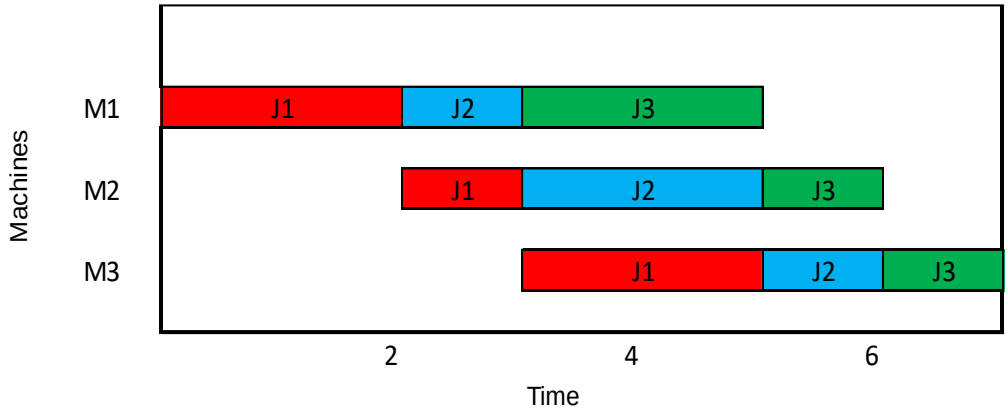


Figura 9. Ejemplo de diagrama de Gantt de un problema de taller de flujo de permutación con restricción no-idle (sin tiempos ociosos entre el procesado de los trabajos). (Fuente: Elaboración propia).

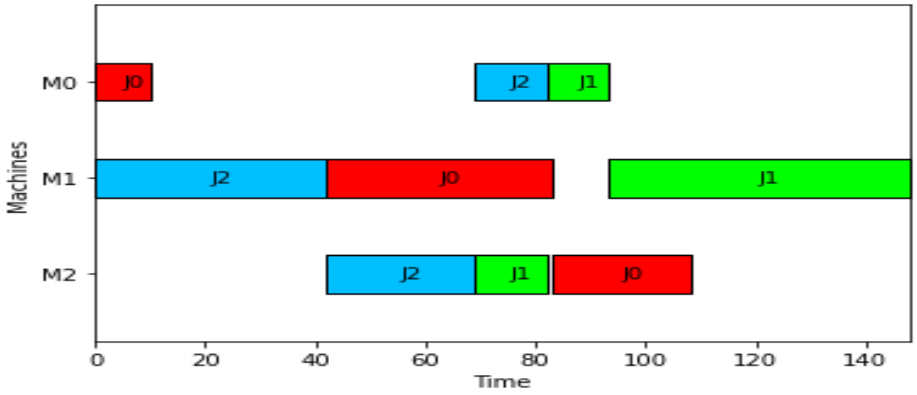


Figura 10. Ejemplo de un diagrama de Gantt para un problema Jobshop donde las rutas imposibilitan el cumplimiento de no-idle. (Fuente: Elaboración propia).

Por tanto, atendiendo al tipo de entorno, restricciones y objetivos seleccionados, se diferencian los siguientes problemas:

1) $J_m \mid \mid$ Cmax. Problema que busca minimizar el máximo tiempo de terminación (makespan).
2) $J_m \mid \mid \sum(W_i CIT_i)$. Problema que busca minimizar la suma total de los tiempos ociosos entre los trabajos en las máquinas. Se recuerda del capítulo anterior que W_i hace referencia al peso asignado a cada máquina i y CIT_i representa el tiempo ocioso (Core Idle Time) en cada máquina i.

Para ilustrar el cálculo de estos dos objetivos, a continuación, se muestra un ejemplo en la Figura 11 de una solución dibujada en un diagrama de Gantt para un problema Jobshop.

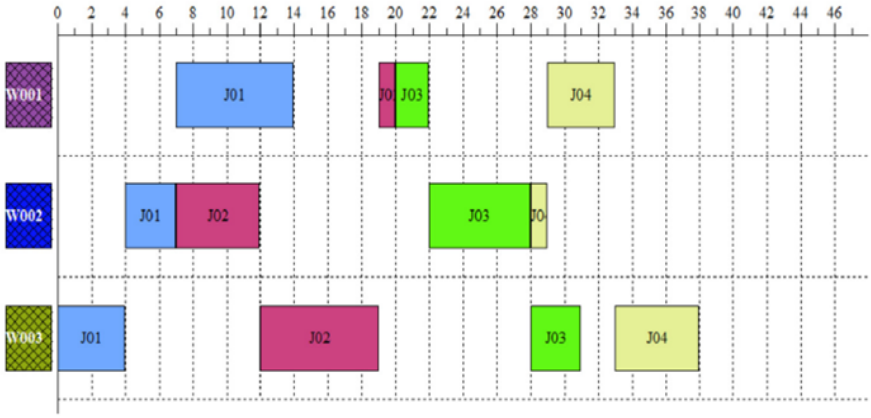


Figura 11. Ejemplo de diagrama de Gantt para un problema Jobshop. [Fuente: Problemas programación de operaciones (Pérez González, Fernández-Viagas, Talens Fayos & Framiñán, 2022)].

El valor de la función objetivo makespan para este ejemplo, resulta inmediato, ya que corresponde con el tiempo de terminación del trabajo J04 en la última máquina de su ruta de procesado W003. El valor de este tiempo es de 38 u.t (unidad de tiempo), por tanto, el valor de makespan, sería de 38.

Para calcular el valor de la función objetivo Core Idle Time para el mismo problema hay que localizar todos los tiempos ociosos para cada máquina y posteriormente sumarlos. En la máquina W001 se aprecia tiempo ocioso entre J01 y J02 (5 u.t) y entre J03 y J04 (7 u.t). La suma de tiempos ociosos para la máquina W001 resulta por tanto de $CIT_{W001} = 12$

En la máquina W002, solo existe tiempo ocioso entre J02 y J03 (10 u.t), por lo que, para esta máquina, la suma de tiempos ociosos resulta de $CIT_{W002} = 10$

Para la última máquina W003, de sumar los tiempos ociosos entre J01 y J02 (8 u.t), entre J02 y J03 (9 u.t) y por último entre J03 y J04 (2 u.t), se obtiene un total de $CIT_{W003} = 19$

Si se estableciese una ponderación para cada máquina, por ejemplo, $W_i = [W_{W001}, W_{W002}, W_{W003}] = [1,3,2]$, esto significaría, a efecto teórico, que la máquina W002 consume tres veces más que la máquina W001. Con estas ponderaciones el sumatorio para hallar el valor de la función objetivo sería:

$$\sum W_i CIT_i = W_{W001} * CIT_{W001} + W_{W002} * CIT_{W002} + W_{W003} * CIT_{W00} = 80$$

3.2 Métodos de resolución propuestos

3.2.1 Marco histórico de aplicación de metaheurísticas

Los primeros estudios realizados en busca de la resolución del JSSP (Job Shop Scheduling Problem) se hicieron a través de algoritmos exactos. Estos algoritmos, a diferencia de los aproximados, son aquellos que pueden asegurar un valor óptimo. Una de las desventajas que surgen con estos algoritmos es que no siempre son aplicables a cualquier problema. A medida que un problema aumenta de tamaño, también se aumenta la complejidad y, por tanto, cuando se tienen recursos limitados de tiempo o memoria, surge la necesidad de buscar otros métodos alternativos. Esto supuso en las últimas décadas un empuje en el estudio de las metaheurísticas (Liang Gao, Xinyu Li, Xiaoyu Wen, Chao Lu, Feng Wen, 2014).

En referencia a los dos tipos de metaheurística de interés en esta literatura, metaheurísticas basadas en trayectoria y metaheurísticas basadas en población, se hace un recorrido de cómo los investigadores a lo largo de los años han desarrollado alguno de estos métodos y cómo se integran para la resolución de problemas de tipo Jobshop.

Respecto a los algoritmos basados en trayectorias, representaron hitos importantes los estudios de los métodos *Simulated Annealing* (Kirkpatrick, Gellat & Vecchi, 1983) y VNS (*Variable Neighbourhood Search*) (Mladenovic y Hansen ,1997).

El método Simulated Annealing (Kirkpatrick, Gellat & Vecchi, 1983) surgió de la inspiración en el proceso de recocido de cerámicas y acero. En estos procesos, el material se somete a incrementos de temperatura que permiten mayor desplazamiento de los átomos y posteriormente a enfriamientos lentos que ayudan a cristalizar. La particularidad que presenta este método es que contiene un parámetro T (Temperatura) que determina la probabilidad de adoptar soluciones alternativas de peor calidad. En ciertos casos donde se adopta una solución peor, se permite la exploración de nuevas trayectorias de búsqueda dando la posibilidad a que los nuevos vecinos generados en el vecindario presenten mejoras con respecto a la mejor solución que fue desestimada inicialmente.

Con el método VNS (Mladenovic y Hansen ,1997) se presentó la idea de utilizar varias estructuras de vecindario que van actualizándose conforme avanza el algoritmo. La utilización de estas estructuras dinámicas, permiten la obtención de vecinos utilizando varios criterios distintos, consiguiendo mayor precisión dada una trayectoria de búsqueda. Este método ha sido utilizado en diversos problemas matemáticos famosos tales como el *Traveling Salesman Problem* (Felipe, Ortuno, & Tirado, 2009) o el *Vehicle Routing Problem* (Kuo & Wang, 2012).

En cuanto a los algoritmos basados en poblaciones, se hace especial mención a los métodos AG (Algoritmo Genético) (Holland, 1975) y PSO (*Particle Swarm Optimization)* (Kennedy y Eberhart, 1995).

La inspiración del Algoritmo Genético (Holland, 1975) tiene su base en la biología y en la evolución genética de los seres vivos. Haciendo analogía con la evolución de las especies, en una población de individuos que generan descendientes, estos sobreviven en un entorno si sus características genéticas así lo permiten. En términos de programación, este algoritmo parte de una población de soluciones a las cuales, se les realizan operaciones de cruce y mutación para la obtención de soluciones descendientes. Estas nuevas soluciones, serán sustituidas en la población si son lo suficientemente buenas una vez evaluadas, es decir, presentan mejores valores en comparación con otras que ya se encuentren dentro de la población.

El método tradicional PSO (Kennedy y Eberhart, 1995), también basa su inspiración en la naturaleza. En este caso, es común encontrar analogías con en el movimiento descrito por organismos vivos como bandada de aves, colonias de insectos, banco de peces etc. Por este motivo, en la descripción del modelo, es común encontrar parámetros definidos como términos físicos como es el caso de partículas, velocidad o desplazamiento. En cuanto a funcionamiento, se parte de una población de soluciones o partículas, a las que se aplican operaciones entre partículas que presentan mejor posición tanto a nivel global como a nivel local. El resultado de estas operaciones es el movimiento de la población hacia una posición mejor.

Para resolver problemas de tipo Jobshop, un gran número de investigadores ha optado por combinar algoritmos basados en trayectorias con algoritmos basados en poblaciones, obteniendo como resultado algoritmos híbridos. El objetivo común por el cual se han desarrollado estos algoritmos es que cada uno presenta ciertos ventajas e inconvenientes. Al aplicar una combinación de ambos, las desventajas pueden corregirse en gran medida. Los algoritmos basados en trayectorias suelen ser precisos realizando evaluaciones de carácter local pero bien es cierto que, si se parte de una solución mala de partida es posible que la solución converja en un mínimo local. Por contraposición, en caso de los algoritmos basados en poblaciones, acota las posibles vías de trayectorias donde encontrar un buen valor a nivel global pero no profundiza tanto a nivel local como se hacía con algoritmos basados en trayectorias. (Liang Gao, Xinyu Li, Xiaoyu Wen, Chao Lu, Feng Wen, 2014).

Teniendo en cuenta que existen numerosas investigaciones, a continuación, se citan casos donde algunos investigadores han hecho uso de algoritmos híbridos a partir de los citados con anterioridad. A modo de ejemplo, se pueden encontrar investigaciones sobre un algoritmo híbrido AG-SA (Chaoyong Zhang, Peigen Li, Yunqing Rao, & Shuxia Li, 2005), sobre un algoritmo PSO-SA (Particle Swarm Optimization y Simulated Annealing) (Niknam, Amiri, Olamaei, & Arefi, 2009), o sobre el algoritmo objeto de estudio de este trabajo PSO-VNS (Particle Swarm Optimization y Variable Neighbourhood Search).(Liang Gao, Xinyu Li, Xiaoyu Wen, Chao Lu, Feng Wen ,2014).

3.2.2 Algoritmo Genético

Antes de describir los pasos a seguir por el Algoritmo Genético, es preciso definir previamente las operaciones de cruce y mutación que van a ser aplicadas. En este caso, se aplican cruce POX y mutación por intercambio:

a) El cruce POX (*Precedence Operation Crossover*) (Zhang, Li, 2008) Es una operación que se realiza entre dos secuencias (padres o *parents*) mediante las que, siguiendo una heurística, se obtienen dos nuevas secuencias (descendientes o *offsprings*). La heurística sigue los siguientes pasos:

- En primer lugar, se dividen aleatoriamente los trabajos presentes en una secuencia en dos subgrupos denominados Jobset1 y Jobset2.
- En segundo lugar, el trabajo que pertenece a Jobset1 es agregado a la misma posición en offspring1 y eliminado en parent1. El trabajo en parent2 que pertenece a Jobset1 se agrega a la misma posición en offspring2 y eliminado en parent2. Finalmente, los trabajos restantes en parent2 se agregan a las posiciones vacías restantes en offspring1 en orden y también se obtiene offspring2 haciendo lo propio con los trabajos restantes de parent1.

Para facilitar la compresión de esta operación, en la Figura 12, se observa un ejemplo de cruce POX entre dos secuencias extendidas para un problema de tipo Jobshop (instancia 3x3). En la imagen se muestra como a partir de las secuencias de partida (parent1 y parent2) se generan las descendientes (offspring1 y offspring2). Para este caso, se ha establecido de forma aleatoria que pertenecen al grupo jobset1 exclusivamente el trabajo 2, mientras que los trabajos restantes (1,3) pertenecen al grupo jobset2

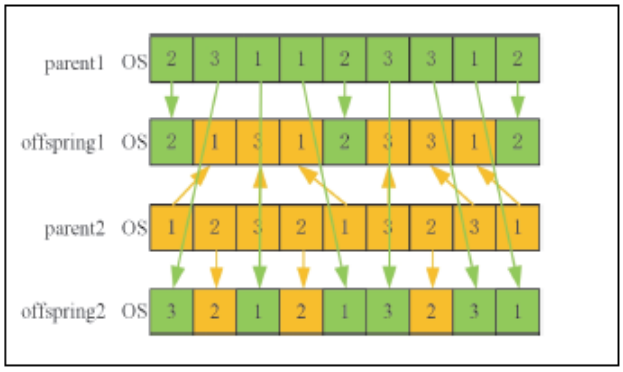


Figura 12. Ejemplo de cruce POX entre dos secuencias extendidas de tipo Jobshop [Fuente: Digital-Twin Based Job Shop Scheduling towards Smart Manufacturing (Yilin Fang, Chao Peng, Ping Lou, Zude Zhou, Jianmin Hu & Junwei Yan, 2019)].

b) La mutación por intercambio es una operación que consiste en la selección aleatoria de dos posiciones de una secuencia con el objetivo de realizar un intercambio de los trabajos correspondientes a estas posiciones. Esta sencilla operación da lugar a una nueva secuencia. (Pérez González, Fernández-Viagas & Framiñán, 2021).

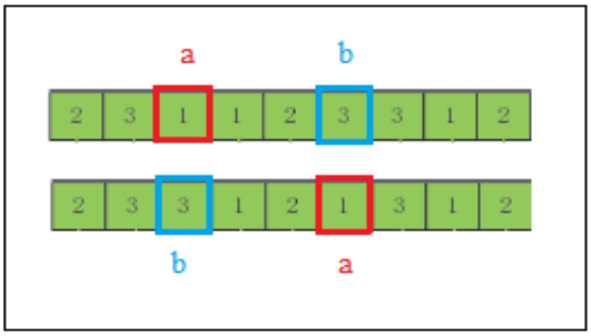


Figura 13. Ejemplo de mutación por intercambio para una secuencia extendida Jobshop [Fuente: Elaboración propia].

Una vez se han definido estas operaciones, el Algoritmo Genético puede describirse siguiendo la siguiente serie de pasos:

1) Se fijan dos parámetros iniciales: Tamaño de población y criterio de parada. El criterio de parada bien puede ser o número de iteraciones o tiempo de ejecución.
2) Se genera una población inicial. En este caso, la población se genera de manera aleatoria.
3) Una vez se tiene una población generada, se evalúan todas las secuencias y se guarda la secuencia con mejor valor resultante de evaluar la función objetivo.
4) Se establece un cruce POX entre dos secuencias padres. En este caso como secuencias padres tomaremos una secuencia aleatoria procedente de la población como parent1 y la secuencia que presenta mejor fitness como parent2. De este cruce, se obtienen dos secuencias descendientes, child1 y child2
5) A los descendientes hallados en el paso anterior, se les realiza una operación de mutación. La operación usada en este caso será la mutación por intercambio, donde dada la secuencia se permutan dos posiciones aleatorias generando así una nueva secuencia. De este paso se obtienen dos nuevas secuencias childmutated1 y childmutated2. La secuencia que presente mejor fitness de entre estas dos será la secuencia candidata a entrar en la población.
6) El mejor childmutated se compara con todas las secuencias presentes en la población. Si esta secuencia presenta mejor fitness que otra secuencia en la población, se intercambiará tantas veces como sea mejor. Si la secuencia childmutated presenta mejor fitness que la mejor secuencia hallada al principio en la población, el valor de la mejor secuencia hallada se actualiza por el de esta nueva secuencia.
7) Al tratarse un proceso iterativo, se comprueba si se cumple la condición de parada, que en este caso es el número de iteraciones fijado al inicio del algoritmo. En caso de no haber finalizado, se realiza de nuevo los mismos pasos desde el paso 4. En caso de finalización, el algoritmo devuelve la secuencia que presenta mejor fitness dentro de la población.

Para un mejor entendimiento de la heurística, en la Figura 14 se presenta el pseudocódigo del algoritmo.

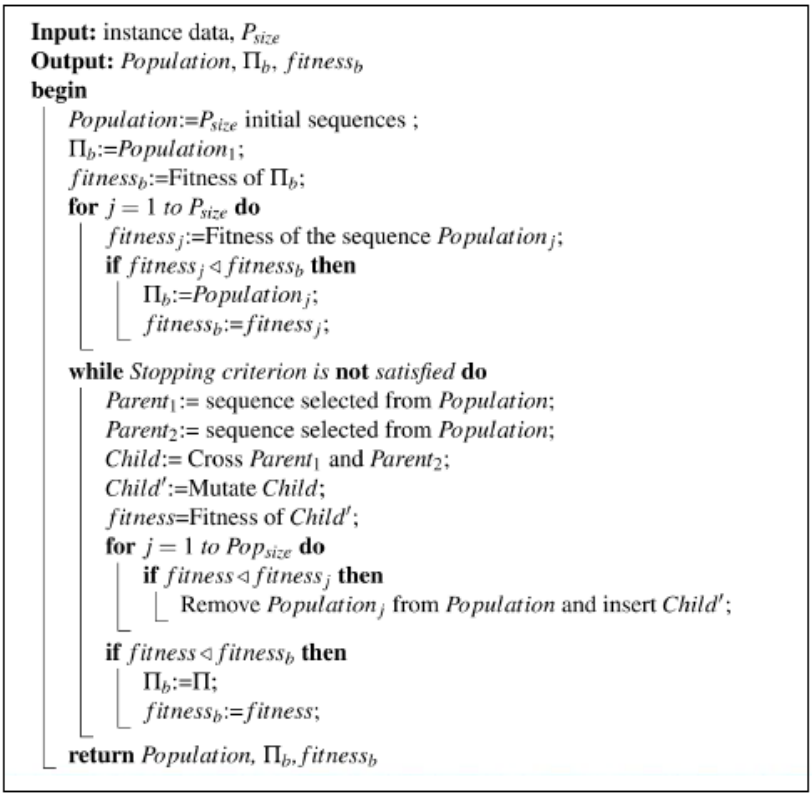


Figura 14. Pseudocódigo del algoritmo Genético. [Fuente: Apuntes de programación de operaciones (Pérez González, Fernández-Viagas & Framiñán, 2021)]

3.2.3 Algoritmo Simulated Annealing

La heurística del algoritmo Simulated Annealing puede describirse en función de la siguiente serie de pasos:

1) En primer lugar, se fijan las variables iniciales temperatura (T) y paso (r). El paso r es un parámetro que se fija al inicio del algoritmo con el objetivo de decrementar el valor de la temperatura. La condición para fijar la r es que sea un valor en el intervalo (0,1) y T>0, con el objetivo de que el parámetro temperatura pueda ir disminuyendo en cada iteración.
2) Se genera una secuencia solución inicial. La secuencia inicial puede generarse siguiendo alguna heurística previa o de manera aleatoria.
3) Proceso iterativo:
   a) Se crea el bucle fijando una condición de parada. Algunos de los criterios más comunes son: fijar un valor máximo alcanzable de temperatura o establecer algún límite de tiempo de ejecución.
   b) Se crea una vecindad dentro del bucle a partir de la secuencia solución. Una vez obtenida la vecindad se compara todas las secuencias vecinas con la propia secuencia. Si alguna secuencia vecina presenta mejor valor que la que se tiene inicialmente se actualiza la secuencia solución por la nueva mejor secuencia encontrada.
   c) En el caso de que no exista ningún vecino mejor que la secuencia solución se abre una posible vía de búsqueda. Esta vía consiste aceptar una secuencia del vecindario con un fitness peor atendiendo a un modelo exponencial de probabilidad donde se tiene en cuenta diferencia de fitness y ciertos valores aleatorios.
   d) Se actualiza el valor de temperatura siguiendo la regla T=T*r. Con ello, la temperatura va decreciendo en cada pasada del bucle.
4) Una vez finalizado el bucle el programa devuelve la mejor secuencia solución.

Para una mejor comprensión de la metodología, en la Figura 15 se muestra el pseudocódigo del algoritmo.

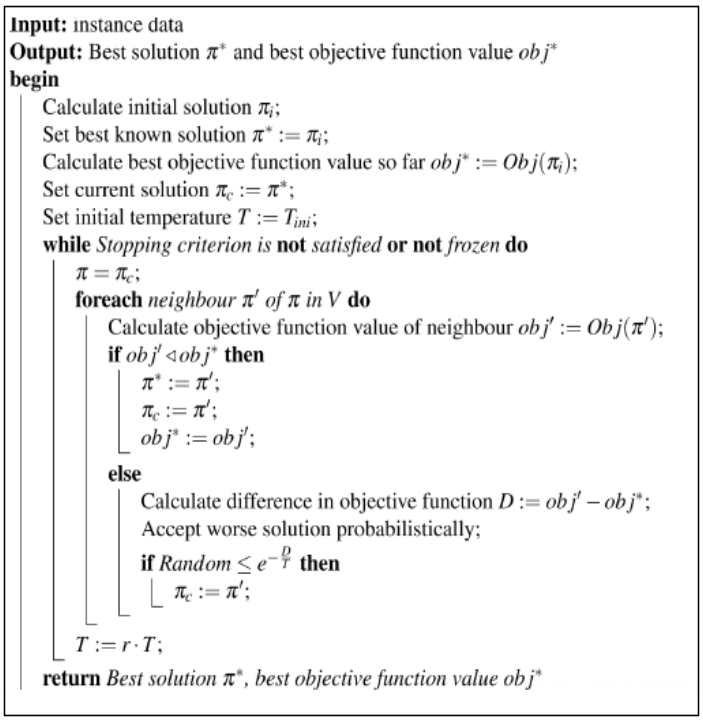


Figura 15. Pseudocódigo del algoritmo Simulated Annealing. [Fuente: Apuntes de programación de operaciones (Pérez González, Fernández-Viagas & Framiñán, 2021)].

3.2.4 Algoritmo Hybrid PSO-VNS (HPV)

En este subapartado, se define el algoritmo Hybrid PSO-VNS tal como se plantea en el artículo "*A hybrid algorithm based on a new neighborhood structure evaluation method for job shop scheduling problem* "(Liang Gao, Xinyu Li, Xiaoyu Wen, Chao Lu, Feng Wen ,2014). A continuación, se muestran los pasos que sigue este método:

1) El primer paso es fijar dos parámetros iniciales: Tamaño de población y criterio de parada. El criterio de parada bien puede ser o número de iteraciones o tiempo de ejecución.
2) Una vez inicializados los parámetros anteriores, se crea aleatoriamente una población del tamaño introducido. Cada secuencia solución de la población se evalúa según la función objetivo y a partir de aquí se generan dos conjuntos:
   a) *Individual extreme library*. Conjunto formado por las tres mejores secuencias dentro de la población inicial.
   b) *Population extreme library*. Conjunto cuyo tamaño corresponde con el 20% del tamaño de la población y está formado por secuencias que presentan distinto valor tras evaluar la función objetivo.
3) Se busca actualizar la población siguiendo una estrategia de búsqueda basada en el método PSO. Previo a describir como se realiza este proceso, es necesario definir la operación de mutación por inserción:
   - La mutación *insertion* consiste en sustraer un elemento de una posición aleatoria de la secuencia para colocarlo en otra posición aleatoria distinta. En la Figura 16 se puede ver un ejemplo donde dada una secuencia se extrae el elemento colocado en la posición 1 (índice 4) y posteriormente se introduce en la posición 2 (índice 6).

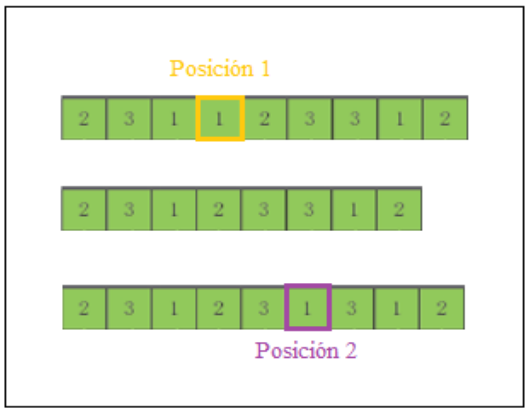


Figura 16. Ejemplo de mutación insertion [Fuente: Elaboración propia].

Definidas todas las operaciones de mutación y cruce que se realizan, la heurística sigue de la siguiente forma:

I. Paso 1: Selección de dos secuencias para establecer una operación de cruce entre ellas. La primera secuencia se obtiene siguiendo el orden de la población y la segunda es extraída aleatoriamente del conjunto Individual extreme library. Ambas secuencias deben cumplir con la condición de que una vez han sido evaluadas según la función objetivo, poseen distinto valor. Una vez seleccionadas, se realiza un cruce entre ambas de tipo POX, del cual se obtienen dos nuevas secuencias.

II. Paso 2: Análogo al paso previo con la salvedad de que la operación de cruce POX se establece entre una secuencia de la población y una secuencia procedente del conjunto Population extreme library. De este paso se obtienen dos nuevas secuencias adicionales.

III. Paso 3: A la secuencia de la población se le realiza una operación de mutación mediante el método insertion. Una vez realizado, se obtiene una secuencia adicional.

IV. Paso 4: Elección de la mejor secuencia obtenida. Tras los pasos previamente realizados, se obtienen cinco nuevas secuencias. La secuencia que evaluada resulte mejor es sustituida por la secuencia de la población. La búsqueda global se terminará cuando se lleve a cabo este proceso de forma ordenada para todas las secuencias de la población.

4) Se aplica el método VNS para cada secuencia de la población actualizada. El diseño del algoritmo, parte de la base de que la solución local encontrada en un vecindario no tiene por qué ser la misma que en otro vecindario diferente. Por ello, la idea principal de este algoritmo es, a partir de una secuencia, generar varias vecindades donde haya posibilidad de encontrar una secuencia mejor que presente mejor fitness que la secuencia de partida. En caso de que se encuentre una secuencia mejor en cualquiera de los vecindarios creados, el proceso se "reiniciará", pero esta vez a partir de la nueva secuencia mejor encontrada, creándose de nuevo las vecindades a partir de esta secuencia. En definitiva, es un proceso iterativo que culmina en el momento que se obtiene una secuencia que no es mejorable en ninguno de los vecindarios creados a partir de ella. El nombre VNS (Variable Neighborhood Search) procede precisamente de que no existen unos vecindarios fijos, sino que van cambiando y se van generando en función de la mejor secuencia que se conozca hasta el momento.

   En este proyecto se estudia el caso que por cada secuencia se generan cinco vecindades distintas. Actualmente existen varias vecindades posibles de aplicación para un problema de JSSP aunque las que se estudian en este documento representan una mezcla de las más eficaces y utilizadas en problemas de este tipo y son las que se utilizan en el artículo base de esta investigación. (Liang Gao, Xinyu Li, Xiaoyu Wen, Chao Lu, Feng Wen ,2014). Las vecindades estudiadas son: Vecindad *N5*, *Random Whole Neighborhood* (RWN), *Random Reverse Neighborhood* (RRN), *Two-Point Exchange Neighborhood* (TEN) y *Random Insert Neighborhood* (RIN).

   a) La vecindad N5 (Nowicki y Smutnicki,1996), genera vecinos teniendo en cuenta caminos y bloques críticos que surgen de analizar el diagrama de Gantt de una solución. Se entiende por camino crítico el camino que une el inicio del procesado con el fin mediante operaciones en las máquinas. Véase en la Figura 17, donde se observa un camino crítico compuesto por el bloque crítico J2, J0 en máquina 1, J1 en máquina 0 y J1 en máquina 2. Haciendo uso de este ejemplo, se entiende como bloque crítico aquel grupo de al menos dos trabajos consecutivos en una misma máquina que forman parte de un camino crítico.

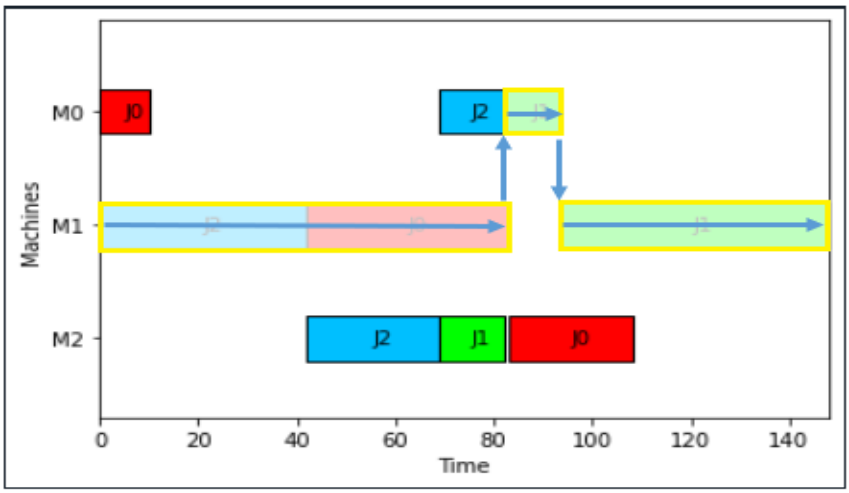


Figura 17. Ejemplo de camino crítico y de bloque crítico [Fuente: Elaboración propia].

Una vez conocidos los términos camino y bloque crítico, la vecindad N5 genera vecinos siguiendo los siguientes pasos:

i. Para el primer bloque crítico permutar las dos últimas operaciones sucesivas.

ii. Para el último bloque crítico, permutar las dos primeras operaciones sucesivas.

iii. Para un bloque intermedio, intercambiar las dos primeras operaciones sucesivas y las dos últimas operaciones sucesivas. En el caso de que el bloque intermedio sea solamente de dos operaciones solo se realizará un intercambio, aunque si dentro del camino crítico existiese otro bloque crítico con mayor número de operaciones, siempre se tomará este para realizar las operaciones y obtener los vecinos.

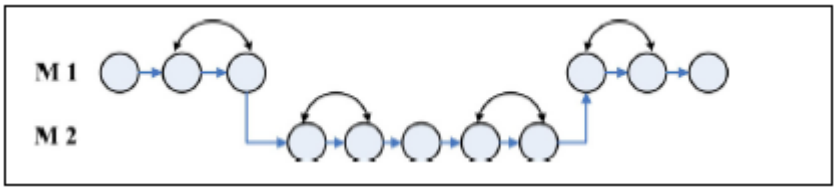


Figura 18. Operaciones realizadas sobre el camino crítico para la obtención de la vecindad N5. [Fuente: A hybrid algorithm based on a new neighborhood structure evaluation method for job shop scheduling problem (Liang Gao, Xinyu Li ,Xiaoyu Wen, Chao Lu & Feng Wen, 2015)].

b) La vecindad Random Whole Neighborhood (RWN) (Cheng, 1997, como se citó en Liang Gao, Xinyu Li ,Xiaoyu Wen, Chao Lu & Feng Wen, 2015) constituye nuevas secuencias vecinas de la permutación de λ elementos diferentes dentro de la secuencia. Este número λ será fijado normalmente en función de la complejidad del problema, variando entre valores de entre 3 y 5. Se suele fijar entre λ=3 y λ=4 cuando el problema es sencillo y λ=5 cuando el problema es complejo (secuencias con más de 200 elementos). En la Figura 19 se muestra un ejemplo de, dada una secuencia y fijado λ=3, cómo se obtendrían todos los vecinos a través de todas las permutaciones posibles entre tres posiciones distintas

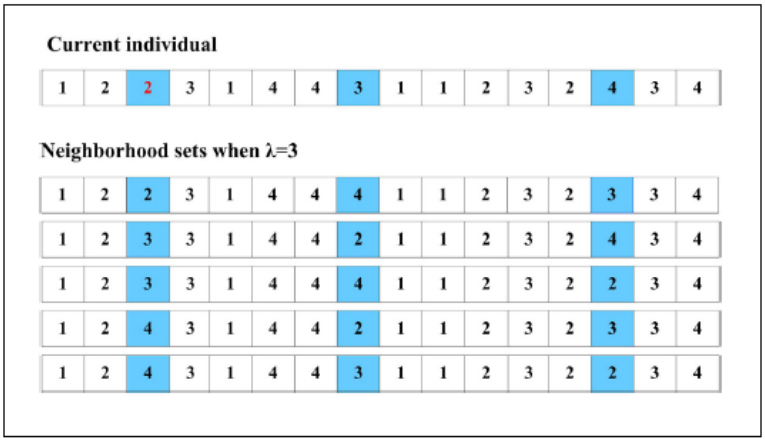


Figura 19. Procedimiento de obtención de vecinos a partir de una secuencia con vecindad RWN para λ=3. [Fuente: A hybrid algorithm based on a new neighborhood structure evaluation method for job shop scheduling problem (Liang Gao, Xinyu Li ,Xiaoyu Wen, Chao Lu & Feng Wen, 2015)].

c) La vecindad Random Reverse Neighborhood (RRN) proporciona nuevas secuencias vecinas a partir de intercambios de dos posiciones aleatorias. Por tanto, cada vecino se diferenciará de la secuencia original en un intercambio de dos posiciones. El número de vecinos a generar se fija previamente.

d) La vecindad Two-Point Exchange Neighborhood (TEN) es más simple que las anteriores ya que solo aporta un vecino formado por el intercambio de dos posiciones aleatorias de la misma manera que se procedía en la vecindad anterior. La diferencia con respecto a la RRN es que en la RRN se tienen múltiples vecinos mientras que en la TEN solo se encontraría un único vecino

e) La vecindad Random Insert Neighborhood (RIN) es similar a la anterior en cuanto a que solo aporta un vecino, con la diferencia de que la secuencia vecina procede de aplicar el método insertion entre dos posiciones en lugar de hacer un intercambio.

Una vez se ha enunciado las cinco estructuras de vecindades. Se define la heurística que sigue el algoritmo VNS mediante los pasos descritos a continuación:

I. Paso 1. Se parte de una población con un conjunto de secuencias soluciones. En este caso, la población no será un conjunto de secuencias aleatorias, sino que recibirá la población actualizada del proceso previo de búsqueda global.

II. Paso 2. Se definen cinco estructuras de vecindades. Las utilizadas son las que se han definido en los subapartados anteriores. En cuanto a notación para entender el proceso, se define cada vecindad como N_k, siendo k un índice entre 1 y 5.

III. Paso 3. Se selecciona s*, secuencia procedente de la población como solución inicial de la búsqueda local.

IV. Paso 4. Buscar un vecino s´ dentro de $s \in N_k(s^*)$ que satisfaga F(s´) <F(s*), siendo F el fitness. Es decir, se busca en el vecindario un vecino que presente mejor fitness que la secuencia original. Si se encuentra un vecino mejor, se actualiza la secuencia s* de la forma s*=s´.

V. Paso 5. El proceso iterativo termina en el momento que no existe en ningún vecindario ninguna secuencia mejor que la que se tiene como solución. En términos de notación si para $\forall s \in N_k(s^*)$ encontramos que F(s) >F(s*), es decir presentan mayor fitness y por tanto peor solución en términos de minimización. En caso de finalizar el proceso iterativo se avanza hasta el paso 6, en caso contrario se vuelve al paso 4.

VI. Paso 6. Una vez actualizada la secuencia en la población se procede de la misma forma con cada una de las secuencias encontradas en la población hasta que todas las secuencias sean actualizadas.

5) Tras la actualización de la población posterior al VNS, se actualizan los dos conjuntos generados en el segundo paso (Individual extreme library y Population extreme library). La actualización consiste en introducir en estos conjuntos la secuencia con mejor valor en la nueva población en reemplazo de la secuencia con peor valor que tuviese cada grupo.

6) En función del criterio de parada fijado en el primer paso, el algoritmo continuará en bucle o se detendrá. En caso de finalización, el algoritmo tomará como solución la mejor secuencia de la nueva población, es decir, la secuencia que presente mejor valor evaluando la función objetivo.

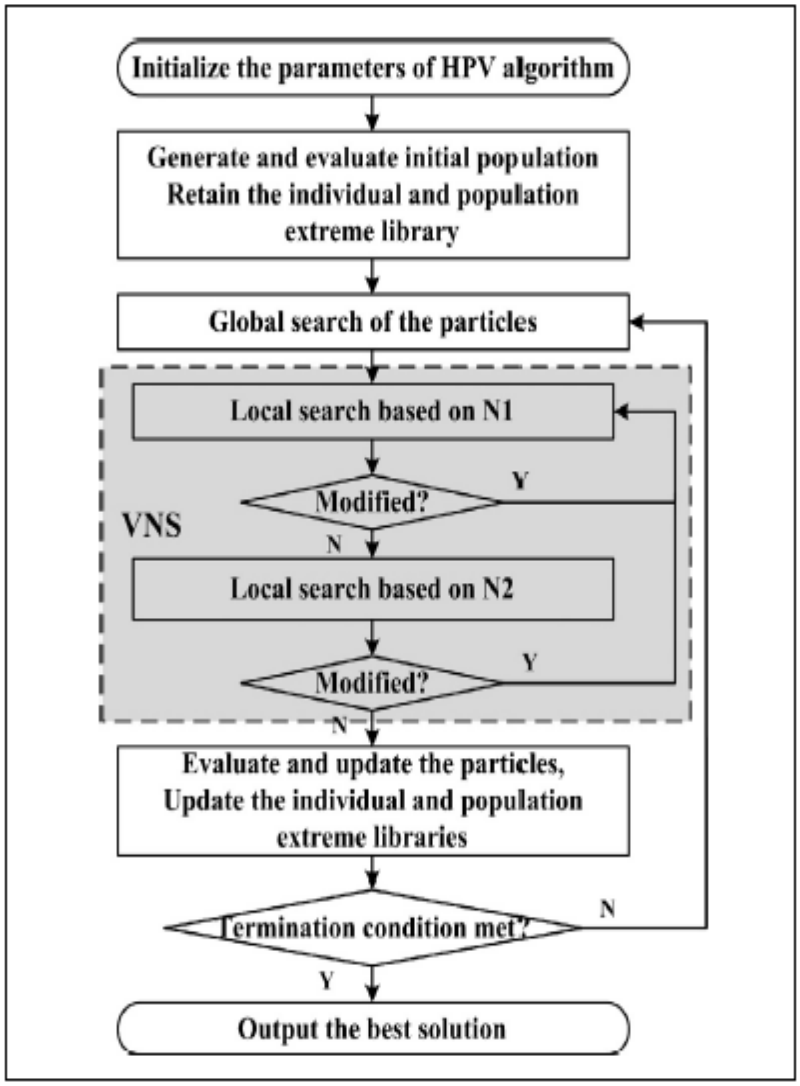


Figura 20. Ejemplo de diagrama de flujo de un algoritmo HPV cuya búsqueda local VNS presenta dos vecindades (N1 y N2). [Fuente: A hybrid algorithm based on a new neighborhood structure evaluation method for job shop scheduling problem (Liang Gao, Xinyu Li ,Xiaoyu Wen, Chao Lu & Feng Wen, 2015)].

4 IMPLEMENTACIÓN DE LOS MÉTODOS

En este cuarto capítulo se muestra cómo se resuelve el problema mediante la codificación en lenguaje Python. Una vez implementados los métodos, se profundiza en el análisis de resultados obtenidos de evaluar las funciones objetivos minimización de makespan y minimización de la suma de los CIT para cada máquina. Dentro de este análisis, se establece una comparativa, a igualdad de tiempo de parada, entre los métodos HPV, Algoritmo Genético y Simulated Annealing. El análisis concluye con una comparativa entre los dos objetivos considerados analizando la relación que existe entre ellos.

4.1 Python y librerías

Para el desarrollo de este trabajo se utiliza el software Spyder como interfaz de programación o IDE (*Entorno de Desarrollo Integrado)*. La codificación en su totalidad se lleva a cabo dentro de este entorno en lenguaje Python. Spyder, desarrollado por Anaconda Inc ©, tiene incluido en su descarga un conjunto de librerías por defecto que usualmente son usadas en el ámbito de la ingeniería y las matemáticas. Numpy, SciPy y MatPlotLib son algunas de estas librerías incluidas en la instalación de Spyder.

Las librerías presentan un conjunto de funciones predefinidas por usuarios con el fin de que otros usuarios puedan descargarlas e implementarlas en su código. En la codificación de este problema se utilizan algunas librerías como *time* (incluye funciones predefinidas para calcular el tiempo de ejecución del programa), *copy* (incluye funciones que permiten copiar listas o arrays en otros) o *random* (funciones que permiten generar un número aleatorio siguiendo unos criterios de generación).

A parte de las librerías citadas, es de especial mención la librería *Scheptk* (Framiñan, 2022). Esta librería está enfocada en la resolución de problemas de *Scheduling*. Entre sus funcionalidades, es de especial interés para este trabajo, poder leer y operar con instancias de tipo Jobshop. Para que la librería sea capaz de leer una instancia de este tipo, se ha de crear un archivo txt con los datos estructurados. Un ejemplo gráfico puede verse en la Figura 21, donde se presenta un archivo txt. con parámetros separados entre corchetes y con los valores separados por comas en caso de vectores y punto y coma en caso de separación de filas de matrices.

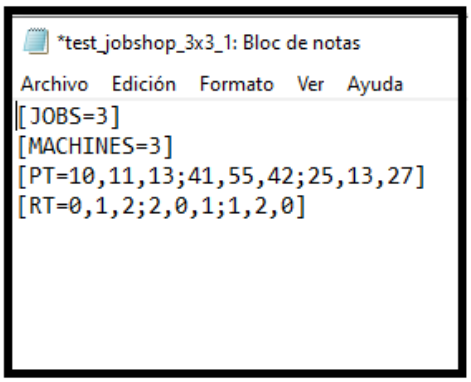


Figura 21. Ejemplo de formato de instancia. Datos en fichero formato txt. [Fuente: Elaboración propia].

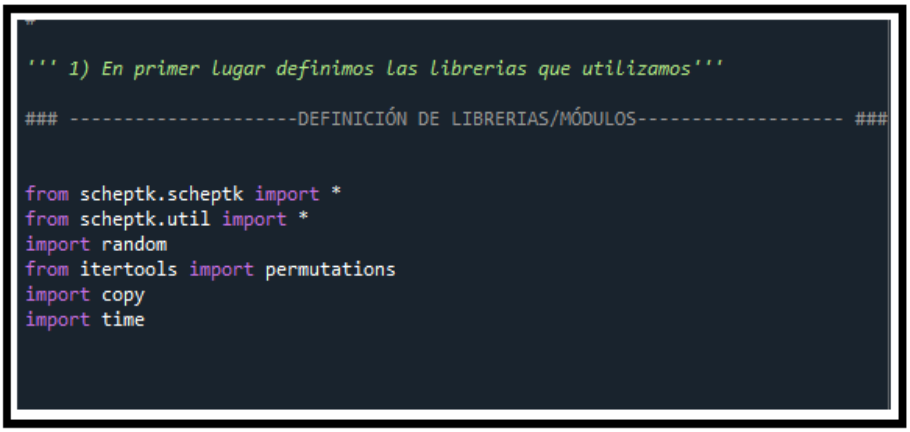
Los archivos de instancias tienen que venir provistos de los datos necesarios para que sean leídos correctamente. De esta forma, el archivo txt. de una instancia de tipo Jobshop, tiene que contener datos de trabajos (JOBS), máquinas (MACHINES) tiempos de proceso (PT) y ruta de trabajo (RT).

En cuanto a aplicación de objetivos, la librería posee un conjunto de funciones objetivos predefinidas. A modo de ejemplo del uso de una de ellas, se tiene la función *instancia.Cmax(secuencia)* que devuelve dada una secuencia el valor de la función objetivo makespan. Otra función de interés es *instancia.print_schedule(secuencia),* con la que se puede obtener la representación del diagrama de Gantt para una secuencia dada.

4.2 Estructura de código

Para los programas realizados con extensión de Python (.py), se sigue el siguiente orden en cuanto a estructura de código:

1) Definición de librerías y módulos. En la cabecera del código se sitúan todos los módulos que son llamados procedentes de una librería. Gracias a esta llamada, somos capaces de introducir en el programa funciones predefinidas externas a nuestro código. Véase Figura 22.



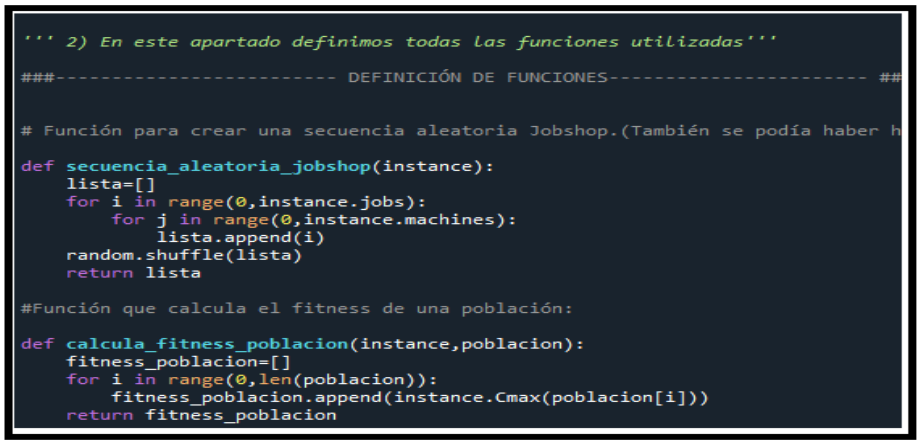
```
''' 1) En primer lugar definimos las librerias que utilizamos'''

### --------------------DEFINICIÓN DE LIBRERIAS/MÓDULOS------------------- ###


from scheptk.scheptk import *
from scheptk.util import *
import random
from itertools import permutations
import copy
import time
```

Figura 22. Definición y llamada de librerías predefinidas. [Fuente: Elaboración propia].

2) Definición de funciones creadas. En este apartado se crean las funciones que son llamadas en la parte operacional del programa o *Main*. En el ejemplo de la Figura 23, se definen las funciones *secuencia_aleatoria_jobshop* y *calcula_fitness_poblacion.*



```
''' 2) En este apartado definimos todas las funciones utilizadas'''

###------------------------ DEFINICIÓN DE FUNCIONES---------------------- ##


# Función para crear una secuencia aleatoria Jobshop.(También se podía haber h

def secuencia_aleatoria_jobshop(instance):
    lista=[]
    for i in range(0,instance.jobs):
        for j in range(0,instance.machines):
            lista.append(i)
    random.shuffle(lista)
    return lista

#Función que calcula el fitness de una población:

def calcula_fitness_poblacion(instance,poblacion):
    fitness_poblacion=[]
    for i in range(0,len(poblacion)):
        fitness_poblacion.append(instance.Cmax(poblacion[i]))
    return fitness_poblacion
```

Figura 23. Definición de funciones creadas en el programa. [Fuente: Elaboración propia].

3) *Main* del programa. Contiene el programa principal donde se realizan las operaciones principales y se hacen las llamadas a las funciones. En la Figura 24 podemos este apartado para el programa HPV.

```
#-------------------------------------------------------------------------------
''' 3) Este apartado equivale al "main", ejecución del algoritmo HPV (Hybrid PSO VNS)'''

### -------------------- MAIN - EJECUCIÓN DE ALGORITMO  ------------------- ###

#         Se establece para ejecutar UNA SOLA INSTANCIA.
#         Bajo las siguientes condiciones:

#       a) La instancia va a ser ejecutada 10 veces.
#       b) Se han fijado las variables Número de población y tiempo de parada a:
#            - Numero de población de 10 individuos --> tamaño_poblacion=10
#            - Tiempo de parada a los 15 segundos --> while(tiempo_ejecucion<15)
#       c) El archivo txt de la instancia ha de estar en el mismo directorio.
#          El nombre de archivo de la instancia es "test_jobshop.txt"

archivo='test_jobshop.txt'
lista_exp=[]

for exp in range(0,10):

    #Inicio de tiempo de ejecución:
    start=time.time()

    #Carga de la instancia Jobshop:
    instancia = JobShop(archivo)

    #Definición del tamaño de la población (population size):
    tamaño_poblacion=10

    #Creación de una población inicial con secuencias random:
    poblacion=creacion_poblacion_inicial(instancia,tamaño_poblacion)
```

Figura 24. Main o ejecución de acciones principales del programa. [Fuente: Elaboración propia].

4) Representación gráfica en forma de diagrama de Gantt. Una vez se tiene la mejor secuencia solución, haciendo uso de la sentencia print_schedule, se plasma la solución en forma de diagrama de Gantt. Figura 25.

```
''' 4) Representación gráfica de la mejor solución en un diagrama de Gantt '''

### -------------------- REPRESENTACIÓN DIAGRAMA DE GANTT  -------------------

instancia.print_schedule(bestseq_pob)
```

Figura 25. Representación gráfica en diagrama de Gantt a partir de la mejor solución obtenida. [Fuente: Elaboración propia].

4.3 Análisis de resultados

4.3.1 Análisis de instancias y medida de comparación

En este apartado se analiza cada algoritmo en base a la experimentación y al tratamiento de los datos obtenidos. Para demostrar la bondad del algoritmo híbrido HPV frente a los algoritmos AG y SA, se prueba una batería de instancias formada por un total de treinta y dos instancias conocidas de tipo Jobshop. Dentro de este conjunto, se incluyen las treinta primeras instancias de Lawrence (LA1 – LA30) (Lawrence, 1984) y las dos instancias de Fisher and Thomson (FT10 y FT20) (Fisher & Thomson,1963).

Se recuerda que las instancias son archivos de extensión txt. donde se recogen los datos necesarios para establecer una secuenciación. Como mínimo, en el caso de un entorno Jobshop, las instancias quedan definidas mediante los siguientes datos: número de trabajos, número de máquinas, ruta de proceso de cada trabajo y tiempo de proceso de cada trabajo en cada máquina.

Lo que se pretende con la evaluación de estas instancias, es poner a prueba las metaheurísticas frente a situaciones distintas y ver cómo estas se comportan. Por este motivo, es interesante analizar instancias con tamaños distintos (mxn), es decir, con número de trabajos(n) y de máquinas(m) variables. Atendiendo al tamaño de las instancias conocidas que se han utilizado para el análisis, se contemplan instancias de varios tamaños (10x5;10x10;15x5;15x10;20x5;20x10). Las treinta y dos instancias de estudio agrupadas por tamaños pueden verse en la Tabla 1.

Tabla 1. Agrupación de instancias por tamaños

	Tamaño de instancias					
	10x5 (5 inst)	15x5 (5 inst)	20x5 (6 inst)	10x10 (6 inst)	15x10 (5 inst)	20x10 (5 inst)
Instancias	LA1; LA2; LA3; LA4; LA5	LA6; LA7; LA8; LA9; LA10	LA11; LA12; LA13; LA14; LA15; FT20	LA16; LA17; LA18; LA19; LA20; FT10	LA21; LA22; LA23; LA24; LA25	LA26; LA27; LA28; LA29; LA30

Para establecer una comparativa, cada instancia va a ser leída en cada uno de los programas realizados para cada algoritmo, evaluándose los dos objetivos de estudio. Una vez cargadas las instancias, en búsqueda de una mayor fiabilidad en los resultados, cada programa se ejecutará diez veces por instancia, de manera que el análisis se realiza con valores promedios en vez de utilizar valores puntuales.

Para ver el desempeño de cada método en función del tiempo, se utiliza el tiempo de ejecución como criterio común de parada para los tres algoritmos. Otro criterio, como puede ser número de iteraciones, no aporta información concreta ya que las iteraciones presentan tiempos de ejecución distintos para cada programa. A modo de ejemplo, una iteración del algoritmo HPV con una población de 200 individuos va a presentar un tiempo de ejecución de algoritmo mucho mayor que una iteración del programa SA que conste de una vecindad sencilla. Por este motivo, para poder establecer una comparativa entre las metaheurísticas, se fija el criterio tiempo de parada. Para todos los programas se ha determinado que el algoritmo pare y arroje una solución una vez alcanzados los 15 segundos para el primer análisis y a 60 segundos para el segundo, para ver el comportamiento conforme aumenta el tiempo.

Una vez se han obtenido los resultados a partir de las pautas anteriores y recopilados en un archivo Excel, se usa como herramienta de comparación el siguiente índice de desviación por cada algoritmo H y para cada instancia K (Ec. 1):

$$RDI_{HK} = \frac{FO_{HK} - FO_K^{MIN}}{FO_K^{MAX} - FO_K^{MIN}} * 100 \ \forall H \ \forall K$$

(Ec. 1)

Este índice se conoce como *Relative Deviation Index* (RDI). Lo interesante de este indicador es que muestra la distancia que existe entre el mejor valor conseguido para una instancia con respecto a los valores obtenidos por cada metaheurística para dicha instancia. Este índice mostrará unos valores comprendidos entre 0 y 1. Cuanto más cercanos sean los valores al mínimo y menor sea la distancia entre valores máximos y mínimos, menor valor de RDI presentará (más cercano a valor 0). En caso contrario, el valor RDI será cercano a valor 1.

Para un análisis más completo, se estudian las instancias agrupadas por tamaños para tener resultados más contrastados de cómo se comportan los algoritmos en función de los tamaños de instancias. Dadas las instancias estudiadas, se establecen los seis grupos de la Tabla 1. (10x5;10x10;15x5;15x10;20x5;20x10).

Por tanto, tras obtener los valores de RDI para todos los algoritmos H y para todas las instancias K, se calcula el ARDI (*Average Relative Deviation Index*) para cada grupo J de instancias que presentan el mismo tamaño. Esto supone, para cada metaheurística, hacer la media de los RDI por grupos de instancias. (Ec. 2).

$$ARDI_{HJ} = \frac{\sum_{K=1}^{N} RDI_{HK}}{N} \quad \forall H \, , \forall K \in J \; con \; J = \{K_1, K_1 \dots, K_N\}$$

(Ec. 2)

4.3.2 Calibración del algoritmo HPV

Previo a la comparativa entre los métodos, se realiza una calibración del algoritmo HPV en función del parámetro población. Esta calibración, consiste en probar cual es el tamaño de población que mejores resultados proporciona para los distintos grupos de instancias. La calibración se realiza para los dos objetivos de estudio y fijando como condición de parada un tiempo de 60 segundos. El tamaño de población que proporcione mejores resultados promedio será el que se utilice para realizar el análisis comparativo de los subapartados posteriores.

Calculados los valores RDI para cada instancia fijando un tamaño de población, se calcula el ARDI para los grupos formados por instancias del mismo tamaño. A partir de los valores ARDI resultantes, se calcula el promedio para todos los ARDI y se obtiene el rendimiento por tamaño de población.

En la Tabla 2, se observa el valor de ARDI, para cada tamaño de instancias y para cada tamaño de población, de resolver el problema makespan con el programa HPV a 60 segundos. Para esta primera situación, el mejor resultado promedio se obtiene cuando el tamaño de población es de P=25.

Tabla 2. Valores de ARDI en función del tamaño de instancia y la población. Calibración del algoritmo HPV aplicando F.O minimización de makespan con tiempo de parada 60 segundos.

	Tamaño de población				
Tamaño de instancia	P=10	P=25	P=50	P=75	P=100
10x5	0,8	0,2659	0,0471	0,0428	0,0822
15x5	0,2	0	0	0	0
20x5	0,1586	0	0,1308	0,0200	0,3333
10x10	1	0,2977	0,1219	0,0961	0,3793
15x10	0,2350	0	0,2866	0,3817	1
20x10	0	0,1727	0,5474	0,7487	1
ARDI promedio por población	0,3989	0,1227	0,1889	0,2149	0,4658

Para la función objetivo minimización de suma de tiempos ociosos entre los trabajos, también se realiza las mismas operaciones, obteniéndose la Tabla 3. En este caso, el mejor tamaño de población encontrado es el de P=10.

Tabla 3. Valores de ARDI en función del tamaño de instancia y la población. Calibración del algoritmo HPV aplicando F.O minimización de Core Idle Time con tiempo de parada 60 segundos.

	Tamaño de población				
Tamaño de instancia	P=10	P=25	P=50	P=75	P=100
10x5	1	0	0,7070	0,7345	0,7877
15x5	0,3354	0,5528	0,2855	0,7853	0,7706
20x5	0	1	0,2256	0,3912	0,5154
10x10	0,0032	1	0,2317	0,4713	0,5106
15x10	0	1	0,5523	0,7228	0,6897
20x10	0	1	0,8315	0,6738	0,6774
ARDI promedio por población	0,2231	0,7588	0,4722	0,6298	0,6586

4.3.3 Calibración de los algoritmos AG y SA

Con el objetivo de obtener un mejor rendimiento de los algoritmos de comparación, se establece una calibración similar a la realizada en el subapartado anterior para los algoritmos AG y SA. En este caso, la calibración para el algoritmo Genético (AG) se realiza, de manera análoga al HPV, variando el parámetro población entre los valores (10,25,50,75,100). En el caso del algoritmo Simulated Annealing (SA), el parámetro variable será el tamaño de la vecindad generada RRN (Random Reverse Neighbourhood), que variará entre los tamaños (50,100,150,200). Ambas calibraciones se realizan a 60 segundos y para las dos funciones objetivo.

Para el algoritmo Genético, los resultados obtenidos para ambos objetivos se recogen en la Tabla 4 y Tabla 5. En este caso, puede verse como el ARDI promedio resulta menor para un tamaño de población de P=10 tanto para makespan como para Core Idle Time.

Respecto al algoritmo Simulated Annealing, los resultados se muestran en la Tabla 6 y la Tabla 7. Para ambos objetivos, el tamaño de vecindad que presenta mejores resultados de ARDI promedio resulta de V=100.

Tabla 4. Valores de ARDI en función del tamaño de instancia y la población. Calibración del algoritmo AG aplicando F.O minimización de Makespan con tiempo de parada 60 segundos.

	Tamaño de población				
Tamaño de instancia	P=10	P=25	P=50	P=75	P=100
10x5	0,1932	0,3751	0,2827	0,4728	0,3823
15x5	0,0000	0,3565	0,3421	0,1632	0,2870
20x5	0,0246	0,1185	0,3333	0,1667	0,2493
10x10	0,3545	0,2771	0,8944	0,2709	0,6038
15x10	0,5128	0,4408	0,6930	0,6657	0,2846
20x10	0,3446	0,4606	0,5432	0,6484	0,7562
ARDI promedio por población	0,2383	0,3381	0,5148	0,3979	0,4272

Tabla 5. Valores de ARDI en función del tamaño de instancia y la población. Calibración del algoritmo AG aplicando F.O minimización de Core Idle Time con tiempo de parada 60 segundos.

	Tamaño de población				
Tamaño de instancia	P=10	P=25	P=50	P=75	P=100
10x5	0,4359	0,3682	0,6114	0,0999	0,4056
15x5	0,4984	0,2663	0,5256	0,6366	0,6396
20x5	0,1260	0,2820	0,2809	0,5447	0,2199
10x10	0,4608	0,5197	0,5037	0,3685	0,3413
15x10	0,1309	0,2519	0,3162	0,6639	0,8764
20x10	0,1018	0,3212	0,9086	0,7365	0,8445
ARDI promedio por población	0,2923	0,3349	0,5244	0,5083	0,5546

Tabla 6. Valores de ARDI en función del tamaño de instancia y la población. Calibración del algoritmo SA aplicando F.O minimización de Core Idle Time con tiempo de parada 60 segundos.

	Tamaño de población			
Tamaño de instancia	V=50	V=100	V=150	V= 200
10x5	0,0353	0,0171	0,4750	0,3681
15x5	0	0	0	0
20x5	0	0,0686	0,2941	0,2614
10x10	0,3572	0,0297	0,9333	0,3393
15x10	0,2502	0,3376	0,8634	0,4112
20x10	0,1370	0,3074	1,0000	0,3735
ARDI promedio por vecindad	0,1300	0,1267	0,5943	0,2922

Tabla 7. Valores de ARDI en función del tamaño de instancia y la población. Calibración del algoritmo SA aplicando F.O minimización de Core Idle Time con tiempo de parada 60 segundos.

	Tamaño de población			
Tamaño de instancia	V=50	V=100	V=150	V= 200
10x5	0,0749	0,2440	0,4	0,1002
15x5	0	0	0	0
20x5	0	0	0	0,1667
10x10	0,6002	0,2484	0,6083	0,5927
15x10	0,2711	0,6630	0,3134	0,7241
20x10	0,5583	0,2636	0,4553	0,9350
ARDI promedio por vecindad	0,2507	0,2365	0,2962	0,4198

4.3.4 Comparativa de algoritmos para minimización de makespan

En este subapartado se comparan los resultados de los algoritmos estudiados (HPV, AG y SA) con la condición de parada a 15 segundos para la primera comparación y a 60 segundos para la segunda. Para ambos tiempos de parada, se analiza la función objetivo makespan.

A partir de los RDI obtenidos para cada instancia, se calculan los valores ARDI por grupos de instancias, de manera que el análisis muestre qué algoritmo presenta menores desviaciones y proporciona mejores valores en función de los tamaños de las instancias estudiadas.

Tras las calibraciones realizadas para cada algoritmo en los subapartados anteriores, se fijan los parámetros de cada algoritmo. Para el HPV, se establece un tamaño de población de P=25, para el AG un tamaño de población de P=10 y para el SA un tamaño de vecindad de V=100 elementos.

El resultado de realizar la primera comparación a 15 segundos puede verse en función de los valores de ARDI en la Tabla 8 y en la Figura 26.

Tabla 8. Valores de ARDI. Comparativa resolviendo makespan con tiempo de parada a t=15 segundos.

	Metaheurística (F.O makespan)		
Tamaño de instancia	AG	SA	HPV
10x5	0,8	0,0523	0,0169
15x5	1	0,0184	0,0256
20x5	0,5	0	0,0391
10x10	1	0,3142	0
15x10	0,9695	0,5332	0
20x10	0,9732	0,0514	0,5959
ARDI promedio	0,8737	0,1616	0,1129

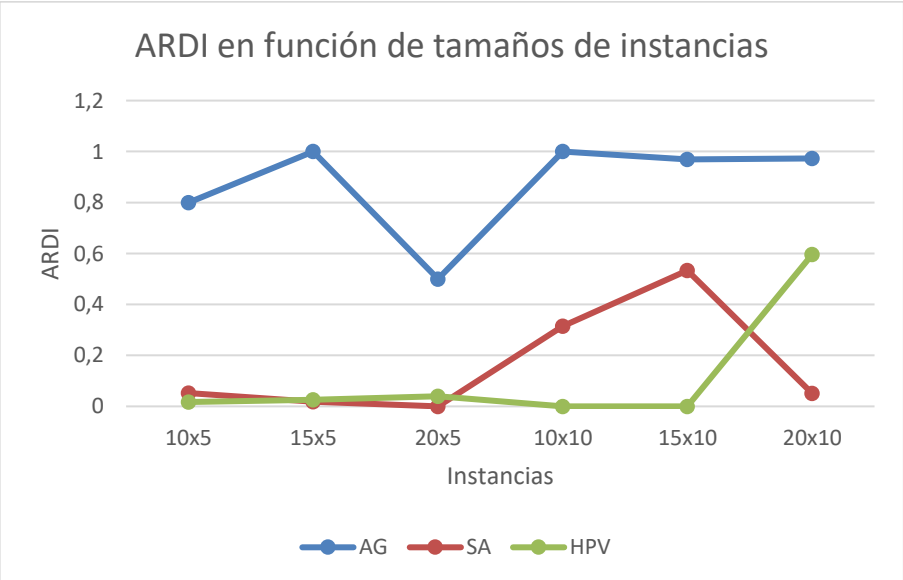


Figura 26. Gráfica comparativa de los valores ARDI resolviendo makespan con parada a t=15 segundos. [Fuente: Elaboración propia].

En esta primera comparativa a 15 segundos, se observa como el algoritmo HPV presenta de media mejores resultados promedios de ARDI que los algoritmos AG y SA. El AG proporciona valores de desviación mayores para todos los tamaños de instancias mientras que, el SA presenta resultados similares al HPV para instancias más pequeñas. Para el tamaño de instancia 20x10, el HPV presenta mayor desviación que el algoritmo SA debido a que, a 15 segundos de tiempo de parada, el algoritmo híbrido no es capaz de realizar suficientes iteraciones dado el tamaño de instancias.

Con respecto a la segunda comparación a 60 segundos, los resultados en función de valores ARDI puede verse en la Tabla 9 y en la Figura 27.

Tabla 9. Valores de ARDI. Comparativa resolviendo makespan con tiempo de parada a t=60 segundos.

	Metaheurística (F.O makespan)		
Tamaño de instancia	AG	SA	HPV
10x5	1	0,1105	0
15x5	0,8	0,0296	0
20x5	0,5	0,0760	0
10x10	1	0,2330	0
15x10	1	0,1288	0,0481
20x10	1	0,0534	0,039
ARDI promedio	0,8833	0,1052	0,0145

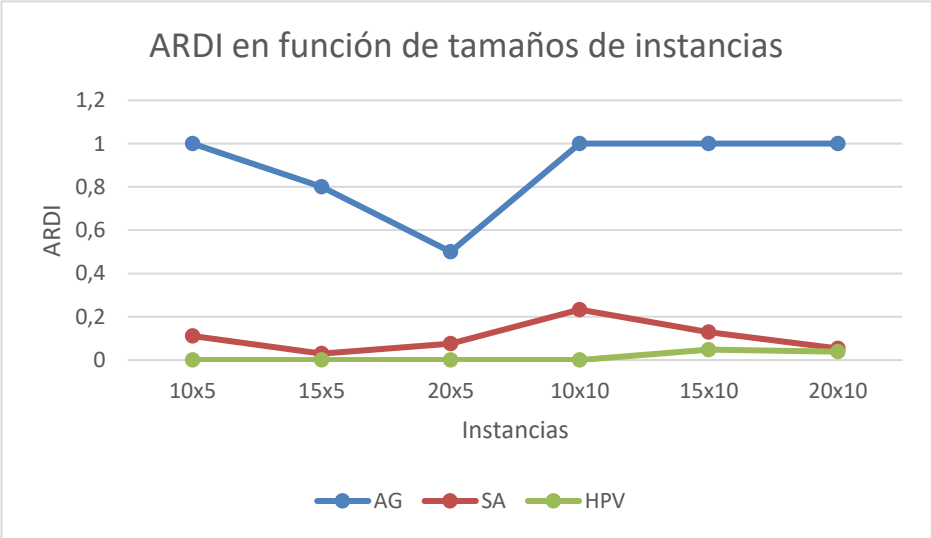


Figura 27. Gráfica comparativa de los valores ARDI resolviendo makespan con tiempo de parada a t=60 segundos. [Fuente: Elaboración propia].

Para esta segunda comparativa, el algoritmo HPV, presenta para todos los tamaños de instancias el mejor resultado de ARDI, mejorando su valor promedio a 0,0145. El método SA también sigue de cerca la tendencia del HPV, mejorando su valor promedio a 0, 1052. Por el contrario, el AG sigue presentando una mayor desviación para todos los tamaños de instancias, manteniendo un ARDI promedio de 0,88.

Tras el análisis con ambos tiempos de parada, se puede indicar que el algoritmo HPV es el que presenta mejores resultados al resolver el problema makespan. El único caso donde el HPV no presenta un buen rendimiento es para un tamaño de instancia de 20x10 con tiempo de parada de 15 segundos (ARDI= 0,5959), dado que al algoritmo no puede realizar suficientes iteraciones. En cuanto se aumenta el tiempo a 60 segundos, esta situación cambia, llegando a convertirse en el método que proporciona menor desviación

para este tamaño de instancia, (ARDI=0,039).

Respecto al rendimiento de los otros métodos, se puede observar que el algoritmo SA presenta buenos valores de ARDI para todos los tamaños de instancias. Para un tiempo de parada de 60 segundos, todos los valores de ARDI son cercanos a 0, obteniéndose el mayor valor para el grupo de instancias de 10x10 (ARDI =0,23). Por el contrario, el AG presenta los valores de desviación más altos para todos los casos resultando ser el que presenta peores resultados de los tres métodos.

4.3.5 Comparativa de algoritmos para minimización de suma Core Idle Time

En este subapartado se comparan, de forma análoga al subapartado anterior, los tres algoritmos (HPV, SA y AG) analizando la función objetivo de suma de tiempos ociosos (suma de CIT).

El análisis también se hace para tiempos de ejecución de 15 y 60 segundos manteniendo los parámetros fijados del apartado anterior para los métodos AG y SA. En el caso del HPV, el tamaño de población de fija en función de los mejores resultados de la calibración realizada, que resulta de P=10 para ambos tiempos de parada. De calcular los ARDI para todos los grupos de instancias, se obtiene el conjunto de resultados de la Tabla 10 y la Figura 28 cuando se fija un tiempo de parada a 15 segundos.

Tabla 10. Valores de ARDI. Comparativa resolviendo suma de Core Idle Time con parada a t=15 segundos.

	Metaheurística (F.O sum CIT)		
Tamaño de instancia	AG	SA	HPV
10x5	0,8409	0,8030	0
15x5	0,5256	1	0
20x5	0,3704	0,9915	0,0438
10x10	0,4952	1	0
15x10	0	1	0,1101
20x10	0	0,9298	0,6270
ARDI promedio	0,3720	0,9540	0,1301

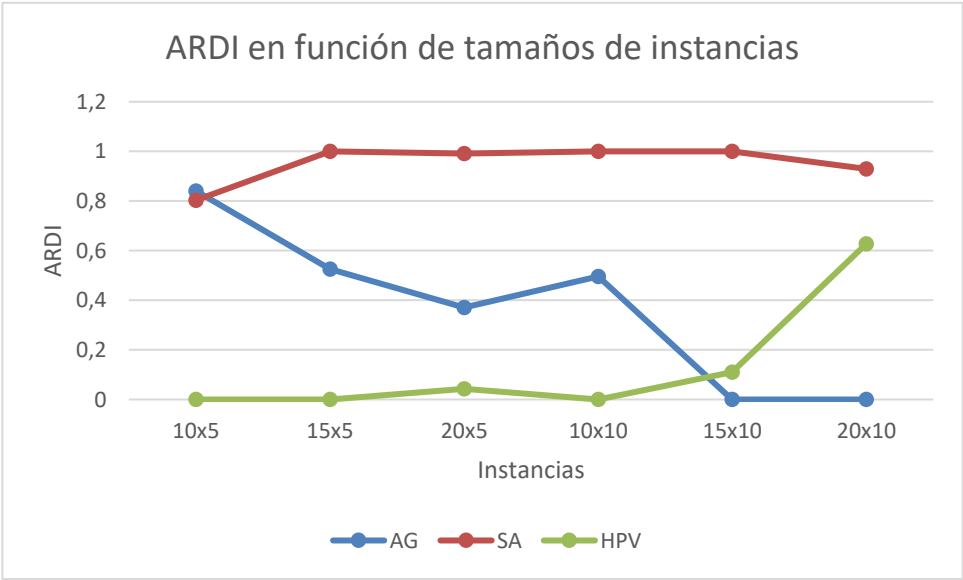


Figura 28. Gráfica comparativa de los valores ARDI resolviendo suma de Core Idle Time con parada a t=15 segundos. [Fuente: Elaboración propia].

El algoritmo HPV, sigue siendo el que presenta menor desviación de media para todas las instancias a excepción de los dos grupos de instancias de mayor tamaño, 15x10 y 20x10, donde el AG consigue alcanzar mejores resultados. A diferencia de los resultados obtenidos para el makespan, el AG mejora positivamente en cuanto a valores medios de desviación mientras que el SA empeora drásticamente su ARDI promedio alcanzando un valor cercano a 1.

Para un tiempo de parada de 60 segundos, se obtienen los datos recogidos en la Tabla 11 y en la Figura 29, para los tres algoritmos:

Tabla 11. Valores de ARDI. Comparativa resolviendo suma de Core Idle Time con parada a t=60 segundos

	Metaheurística (F.O sum CIT)		
Tamaño de instancia	AG	SA	HPV
10x5	1	0,3375	0,0032
15x5	1	0,1691	0,1341
20x5	0,9401	0,4146	0,0068
10x10	0,8993	0,6487	0
15x10	0,4980	0,9382	0
20x10	0,4915	0,0233	0,8501
ARDI promedio	0,8048	0,4219	0,1657

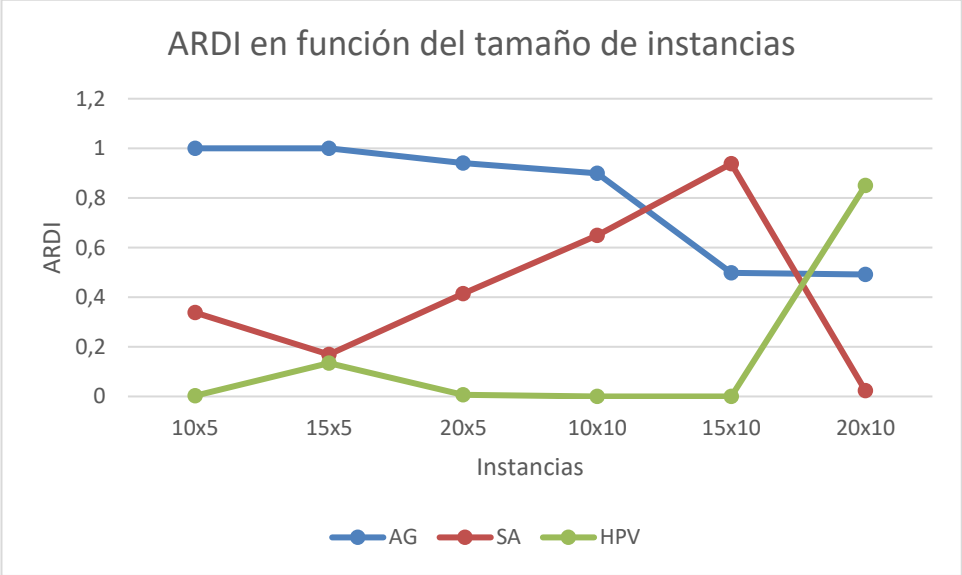


Figura 29. Gráfica comparativa de los valores ARDI resolviendo suma de Core Idle Time con parada a t=60 segundos. [Fuente: Elaboración propia].

En esta comparativa también se sigue manteniendo que el algoritmo híbrido es el que consigue mejores resultados promedios. A diferencia del análisis a 15 segundos, el HPV mejora al algoritmo AG para instancias de tamaño 15x10 mientras que, para las instancias de tamaño 20x10 es el que peor rendimiento proporciona.

A mayor tiempo de parada, los algoritmos AG y SA cambian la tendencia. El AG presenta un valor de ARDI promedio de 0,8 mientras que el SA presenta un valor de 0,42, siendo este último el que mejor ARDI obtiene para el grupo de instancias de mayor tamaño 20x10.

A modo conclusión de esta comparativa, el algoritmo HPV presenta también mejores resultados tras cambiar la función objetivo a suma de Core Idle Time. Tras analizar las desviaciones de este método en función de los tamaños de instancias, en la mayoría de los tamaños se obtienen valores de ARDI cercanos a 0, menos para el tamaño de instancia 20x10, donde el valor de ARDI crece.

Con respecto a los algoritmos AG y SA, los resultados dependen del tiempo de parada que se aplique. Para un tiempo de parada de 15 segundos, el algoritmo AG resulta mejor opción que el SA mientras que para un tiempo mayor de 60 segundos es el SA el que presenta mejor ARDI promedio frente al AG.

4.3.6 Relación entre objetivos: makespan y suma de Core Idle Time

Para finalizar con el análisis de resultados, es de interés mostrar la relación existente entre los dos objetivos de estudio makespan y Suma de CIT. Hasta el momento, se ha analizado cada función objetivo por separado, por lo que, en este subapartado, el propósito es demostrar cuánto de buena es una solución para un objetivo que no se está minimizando directamente. Para ver esta relación, se analizan las desviaciones entre valores para un objetivo dados de minimizar dicho objetivo con respecto a los valores de minimizar el objetivo contrario. Este estudio se realiza para los datos obtenidos aplicando el algoritmo HPV con tiempo de parada 60 segundos, por grupos de instancias.

Para el cálculo de la desviación, se utiliza la desviación típica (σ). Atendiendo a la fórmula (Ec. 3), $S_{OBJ1-OBJ2}$ representa el valor de evaluar un objetivo en una solución obtenida de minimizar el otro objetivo y $S_{OBJ1-OBJ1}$ representa el valor de evaluar un objetivo en una solución resultante de minimizar el propio objetivo. Como cada instancia es ejecutada diez veces, estos valores representan valores medios de soluciones para cada instancia y no valores de soluciones puntuales. La desviación es calculada para cada grupo de instancia J. Los resultados de estos valores de desviación pueden verse en la Tabla 12 y en la gráfica de la Figura 30.

$$\sigma_J = \sqrt{\frac{\sum_{k=1}^{N} (S_{OBJ1-OBJ2} - S_{OBJ1-OBJ1})^2}{N}} \quad \forall K \in J \; con \; J = \{K_1, K_1 \dots, K_N\}$$

(Ec. 3)

Tabla 12. Valores de desviación típica entre valores dados para un objetivo de la aplicación del otro objetivo. Clasificados para grupos de instancias.

	Desviación típica (σ)	
Tamaño de instancia	makespan	suma CIT
10x5	63,0169	244,1688
15x5	36,5963	352,6568
20x5	154,4391	511,9524
10x10	66,2043	486,4239
15x10	78,0734	311,1048
20x10	104,1708	305,6213
σ promedio	83,7501	368,6547

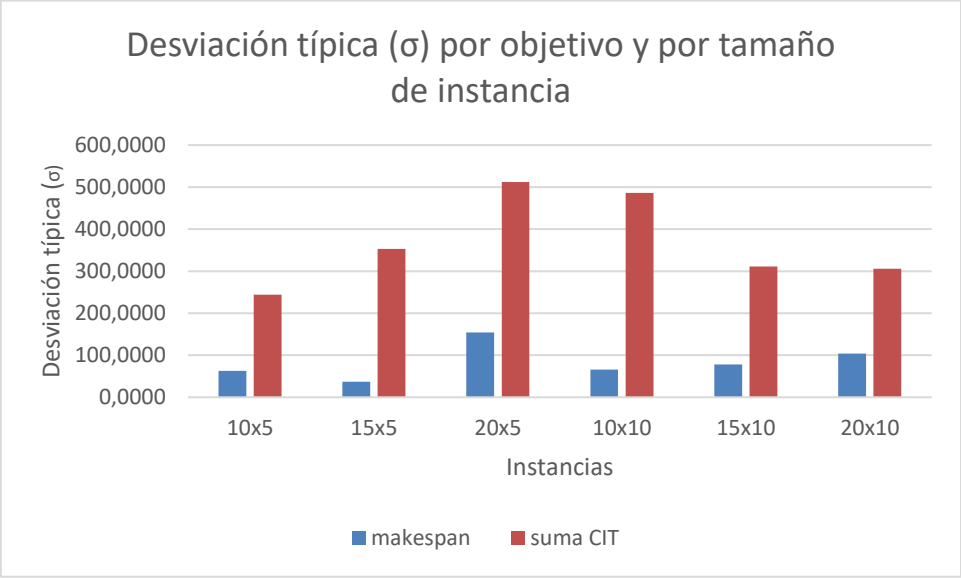


Figura 30. Gráfica de los valores de desviación típica para cada objetivo y por tamaño de instancias. [Fuente: Elaboración propia].

A la vista de los resultados obtenidos, en caso de aplicar la función objetivo minimización de la suma de CIT, se obtiene menor desviación de valores promedios de makespan que en caso contrario (valores promedios de suma de CIT de aplicar función objetivo makespan). Con estos resultados se llega a la conclusión de que evaluando la función objetivo suma de tiempos ociosos (CIT) se minimiza en mayor medida el makespan que lo que se llega a minimizar la suma de CIT cuando lo que se evalúa es la función objetivo makespan. Esto conduce a aconsejar el uso de la función objetivo suma de CIT por delante de makespan siempre y cuando no se tenga un objetivo claramente definido.

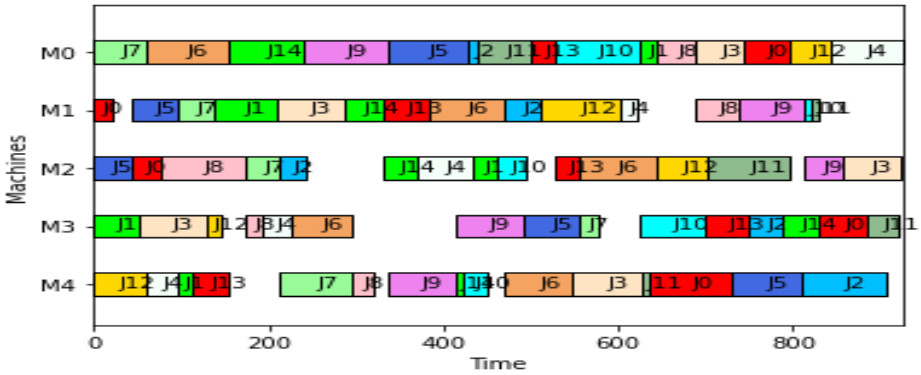


Figura 31. Ejemplo de solución dada por el método HPV para la instancia LA06 (15x5) evaluando makespan[Fuente: Elaboración propia].

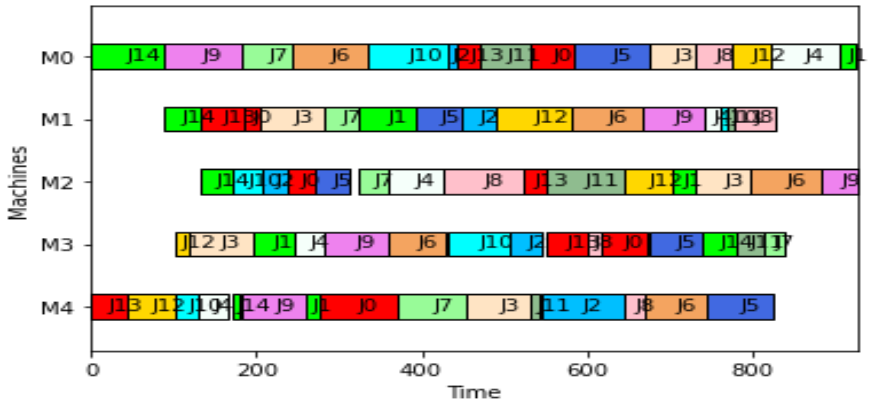


Figura 32. Ejemplo de solución dada por el método HPV para la instancia LA06 (15x5) evaluando suma de CIT. [Fuente: Elaboración propia].

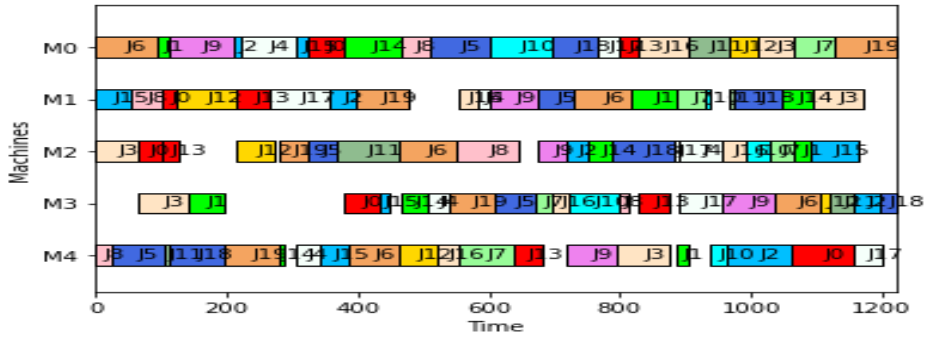


Figura 33. Ejemplo de solución dada por el método HPV para la instancia LA11 (20x5) evaluando makespan. [Fuente: Elaboración propia].

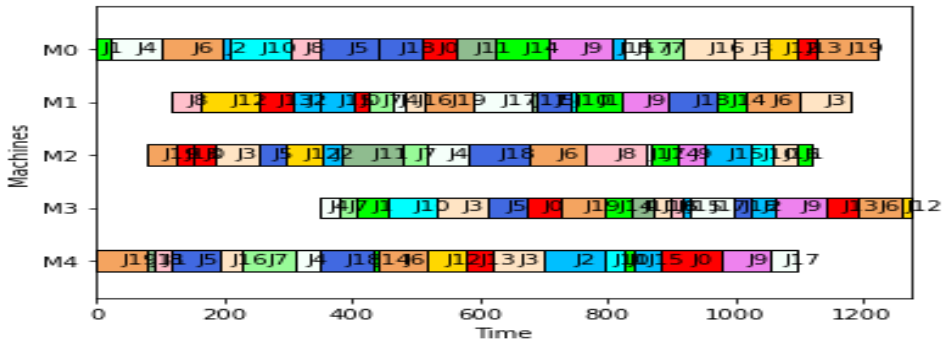


Figura 34. Ejemplo de solución dada por el método HPV para la instancia LA11 (20x5) evaluando suma de CIT. [Fuente: Elaboración propia].

5 CONCLUSIONES

Para finalizar, en este quinto y último capítulo se exponen las principales conclusiones del estudio realizado, haciendo recorrido desde el punto de partida de la investigación hasta la obtención de resultados. También se proponen algunas líneas de trabajo futuro en base a lo que ha sido tratado a lo largo de este proyecto.

5.1 Conclusiones

La preocupación por la sostenibilidad lleva a estudiar como el malgasto energético puede afectar al desarrollo sostenible tanto a nivel ambiental como a nivel económico. Por ello, de la relación que existe entre una producción sostenible y la eficiencia, se ha estudiado desde la programación de operaciones como minimizar el consumo energético. El consumo extra que supone de mantener una producción durante más tiempo del que debiese o, el que supone de tener intervalos de tiempos ociosos en las máquinas, puede ser corregido a partir de un buen programa de producción. Con este propósito, se ha estudiado un problema de tipo Jobshop donde se han aplicado las funciones objetivo: minimización de la suma de Core Idle Time (CIT) y minimización de makespan.

Tras analizar la relación entre los valores obtenidos de evaluar ambos objetivos para este tipo de problema, se llega a la conclusión de que cuando se busca minimizar la suma de CIT se llega también a una buena solución de makespan. En caso contrario, si lo que se busca es minimizar el makespan, existe mayor desviación de valores de suma de CIT. Por esta razón, de aplicar a un caso real, sería más apropiado, en el mayor de los casos, minimizar la suma de CIT ya que se tendría en cuenta los picos de consumo de arranque de las máquinas y, además, se lograría alcanzar un valor de makespan muy bueno. En caso de que los tiempos ociosos supongan un consumo despreciable o significativamente menor frente a mantener una planta en funcionamiento, sería aconsejable minimizar el makespan.

En cuanto a métodos de resolución, en este trabajo se ha profundizado en la resolución mediante métodos aproximados o metaheurísticas, enfocando el desarrollo en el algoritmo híbido Hybrid PSO-VNS (HPV). La ventaja de ser un híbrido entre dos metaheurísticas distintas, es clave para la obtención de una buena solución ya que reúne heurísticas propias de algoritmos basados en población y de algoritmos basados en trayectorias. El algoritmo, a partir de una población con varias soluciones distintas, es capaz de acotar rápidamente una solución de forma similar a cómo funciona el algoritmo PSO. De manera análoga al método VNS, la aplicación de varias estructuras de vecindario ayuda a profundizar en posibles caminos de mejora.

Con respecto a la comparativa analítica entre el algoritmo HPV y los métodos Algoritmo Genético (AG) y Simulated Annealing (SA), se pueden sacar varias conclusiones. Tras la obtención de los valores de ARDI por conjunto de instancias, se determina que el HPV presenta mayor fiabilidad en cuanto a valores promedios de desviación. También se podría decir que el hecho de aumentar el tiempo de condición de parada ayuda a que el algoritmo realice un mayor número de iteraciones y de media proporcione mejores resultados y menor desviación. Esto se puede comprobar viendo cómo se mejoran los resultados cambiando el tiempo de condición de parada de t=15 segundos a t=60 segundos. La calibración de los algoritmos, en base a alguno de sus parámetros, también puede ser diferencial en los resultados. Por ello, de cara al análisis comparativo se ha calibrado el HPV en función de la población para la obtención de mejores resultados.

Respecto a la implementación del código, el uso de IDE Spyder y lenguaje Python resulta una buena herramienta completa, intuitiva y compacta gracias al uso de librerías y módulos predefinidos. Bien es cierto que, la principal desventaja de usar esta combinación para programar es que pueda existir la necesidad de más memoria para la ejecución del código a diferencia de la que usarían otros lenguajes, como por ejemplo C++, aunque el tiempo de ejecución, la memoria usada y la bondad del programa van a depender de muchos más factores.

5.2 Línea de trabajo futuro

Una vez realizado este estudio, se proponen algunas líneas de trabajo futuro. Una de ellas pudiera ser investigar más metaheurísticas de tipo híbrido y establecer comparativas con este algoritmo HPV. Es posible que existan otros métodos donde, aplicando otras operaciones de cruce u otras vecindades, se llegue a algoritmos con mejores resultados.

Para el caso de realizar otro análisis con algoritmos distintos, sería conveniente probar una batería más grande de instancias de mayor tamaño y establecer comparativa fijando como condición de parada un tiempo que supere varios minutos para evaluar resultados distintos y ver cómo se comportan los distintos tamaños de población.

La versatilidad de Python permite crear nuevas librerías o adoptar librerías ya existentes para la mejora del código, por lo que otra línea de posible trabajo en el futuro podría ser la creación de nuevos módulos que ayuden a la programación. Dentro de esta línea, pensando en el uso por parte de un usuario final, también se propone la creación de algún tipo de interfaz de usuario que mejore la experiencia, de manera que fuese accesible para cualquier usuario y no hubiese necesidad de modificar el código por parte del programador.

En cuanto a aplicación, sería interesante una situación real donde se tengan datos del consumo energético de una instalación y se pudiera establecer una ecuación de consumo en kW. En ese caso, cabría la posibilidad de realizar un análisis teniendo en cuenta los valores de los pesos de las máquinas para el caso de función objetivo CIT.

BIBLIOGRAFÍA

Felipe, A., Ortuno, M. T., & Tirado, G. (2009). The double traveling salesman problem with multiple stacks: A variable neighborhood search approach. Computers & Operations Research, 36(11), 2983–2993.

Fisher, H and Thompson, G.L. (1963) *Probabilistic learning combinations of local job-shop scheduling rules*. Industrial Scheduling, vol. 1, no. 2, 225–251.

Framiñan, J.M (2022). Repositorio para instalación de librería Scheptk. Consultada el 27 de Julio de 2024. (https://github.com/framinan/scheptk).

Framiñán JM, Leisten R y Ruiz R (2014) "Manufacturing Scheduling Systems: An interated view on Models, Methods and Tools". Springer.

Gao, L., Peng, C. Y., Zhou, C., & Li, P. G. (2006). Solving flexible job-shop scheduling problem using general particle swarm optimization. In The 36th CIE Conference on Computers Industrial Engineering (pp. 3018–3027).

Graham, R. L., Lawler, E. L., Lenstra, J. K., & Kan, A. H. G. R. (1979). *Optimization and heuristic in deterministic sequencing and scheduling*: a survey. Annals of Discrete Mathematics, 5, 287–326.

Holland, J. H. (1975). *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. Univeristy of Michigan Press

Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. In: Proceedings of IEEE International Conference on Neutral Networks, Australia, Perth (pp. 1942–1948).

Kirkpatrick, S., Gellat, C.D, Vecchi, M.P (1983). *Optimization by simulated annealing*. Science, 220, 671-680.

Kuo, Y., & Wang, C. (2012). A variable neighborhood search for the multi-depot vehicle routing problem with loading cost. Expert Systems with Applications, 39 (8), 6949–6954.

Lawrence, S. (1984). *Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques (Supplement)*. Technical report, Graduate School of Industrial Administration, Carnegie Mellon University. Pennsylvania.

Liang Gao, Xinyu Li, Xiaoyu Wen, Chao Lu, Feng Wen (2014). *A hybrid algorithm based on a new neighborhood structure evaluation method for job shop scheduling problem*. Computers & Industrial Engineering 88, 417-429.

Naciones Unidas (2015). *Transforming our world: the 2030 Agenda for sustainable Development*. Resolution adopted by the General Assembly on 25 September 2015.

Nowicki, E., & Smutnicki, C. (1996). *A fast taboo search algorithm for the job shop problem*. Management Science, 42, 797–813.

Melián, B., Moreno Pérez, J.A, Moreno Vega, J.M. (2003). *Metaheurísticas: una visión global*. Revista Iberoamericana de Inteligencia Artificial 19, 7-28.

Mladenovic, N. & Hansen, P. (1997). *Variable Neighbourhood Search*. Computers & Operations Research 24, 1097-1100.

Perez González, P., Framiñán Torres, J.M, Fernández-Viagas, V., Talens Fayos, C. (2022) *Apuntes de Programación de Operaciones*. Escuela Técnica Superior de Ingeniería de Sevilla. Universidad de Sevilla.

Spencer, C. (2018). Repositorio para instalación de librería Critical Path. Consultada el 27 de Julio de 2024 (https://pypi.org/project/criticalpath/).

World Commision on Environment And Development (WCED) (1987): *Our Common Future* (Brundtlland Report), United Nations.

Zhang, C. Y., Rao, Y. Q., & Li, P. G. (2008). An effective hybrid genetic algorithm for the job shop scheduling problem. International Journal of Advanced Manufacturing Technology, 39, 965–974.

Zhang, C. Y., Li, P. G., Rao, Y. Q., & Guan, Z. L. (2008). A very fast TS/SA algorithm for the job shop scheduling problem. Computers and Operations Research, 35(1), 282–294.

Zhang, C. Y., Li, P. G., Rao, Y. Q., & Li, S. (2005). A new hybrid GA/SA algorithm for the job shop scheduling problem. Lecture Notes in Computer Science, 246–259.

© 2024 Anaconda Inc. All rights reserved. Enlace de instalación (Python + entorno de desarrollo IDE*, Spyder). Consultada el 27 de Julio de 2024. (https://www.anaconda.com/download/).

ANEXO. CÓDIGO EN PYTHON

En este anexo se recopilan los programas realizados en código Python para los tres algoritmos Hybrid PSO-VNS (HPV), Algoritmo Genético (AG) y Simulated Annealing (SA).

Para la evaluación de los objetivos, se define la función "*FUNCION_OBJETIVO (instancia,secuencia)"*. Dada una instancia y una secuencia, esta función devuelve el valor de la evaluación de la suma de tiempos ociosos (CIT). Para la evaluación de makespan, se puede hacer a través de la función predefinida en la librería Scheptk como "*instancia.Cmax(secuencia)"*.

En caso de que cada máquina tuviese un consumo distinto, se ha definido un vector de pesos para definir la importancia correspondiente a cada máquina. Los pesos han sido definidos con valor 1, ya que para este estudio se ha considerado el mismo consumo para todas las máquinas. De ser de interés el asignar pesos a las máquinas, se modificarían los componentes del vector *pesos_maquinas* dentro de la función *FUNCION_OBJETIVO*.

Para el funcionamiento del siguiente código, es imprescindible tener las librerías instaladas. En caso de no tener las librerías, en la bibliografía del presente documento se incluyen los enlaces para la instalación de las que han sido utilizadas.

Para enlazar las instancias con el programa, el archivo de extensión .PY ha de encontrarse en el mismo directorio que la batería de instancias. El nombre de los archivos txt. de las instancias han de ser para las instancias FT10 y FT20 respectivamente *"test_jobshop_ft1.txt"* y "*test_jobshop_ft2*". Para las 30 primeras instancias de Lawrence, se nombran como "*test_jobshop_LA1.txt"*, "*test_jobshop_LA2.txt"*, "*test_jobshop_LA3.txt"* … hasta "*test_jobshop_LA30.txt"*. El programa, una vez finaliza la ejecución, genera un fichero txt. donde se recogen los resultados obtenidos.

A.1 Código HPV

```
#-------------------------------------------------------------------------------

''' 1) En primer lugar se definen las librerias '''

### ---------------------DEFINICIÓN DE LIBRERIAS/MÓDULOS------------------- ###

from scheptk.scheptk import *
from scheptk.util import *
import random
from itertools import permutations
import copy
from criticalpath import Node
import time

#-------------------------------------------------------------------------------

''' 2) En este apartado se definen todas las funciones '''

###------------------------ DEFINICIÓN DE FUNCIONES---------------------- ###

#Función que devuelve el valor de la función objetivo
# F.O --> MINIMIZAR TIEMPO OCIOSO EN LAS MAQUINAS (PONDERADO)

def FUNCION_OBJETIVO(instancia,secuencia):
```

```
#Funcion objetivo - MIN CORE IDLE TIME PONDERADO (máquinas con distinto peso).
#Como las máquinas tienen distinto peso, en primer lugar vamos a asignar los pesos a las
#máquinas en el vector pesos_maquinas=[]:
#Si no se quiere tener en cuenta la ponderación, poner todos los pesos=1
######################

pesos_maquinas=[1,1,1,1,1,1,1,1,1,1]

######################

#Obtencion del orden de los nuevos indices.Tras las siguientes operaciones se obtiene
#una lista li_ind[] que contiene los trabajos en cada maquina en el orden que son procesados
#Para ello tambien se ha ordenado sus ct y sus tiempos de proceso (ct_nuevo[] y pt_nuevo[]).
#En este apartado también se ha calculado los tiempos de inicio de cada trabajo(ci_nuevo[])

ct, job_order = instancia.ct(secuencia)
tiempos=instancia.pt
m=copy.deepcopy(ct)
s=[]
li=[]
li_ct=[]
li_ind=[]
li_temp=[]
for j in range(0,len(m)):
    for i in range(0,len(m[j])):
          li.append([m[j][i],i,tiempos[j][i]])
    li.sort()
    sort_ct=[]
    sort_index = []
    sort_temp=[]
    for x in li:
          sort_ct.append(x[0])
          sort_index.append(x[1])
          sort_temp.append(x[2])
    li_ct.append(sort_ct)
    li_ind.append(sort_index)
    li_temp.append(sort_temp)
    li=[]
ct_nuevo=copy.deepcopy(li_ct)
pt_nuevo=copy.deepcopy(li_temp)
ci_nuevo=[]
for k in range(0,instancia.machines):
    s=[(i-j) for i,j in zip(ct_nuevo[k],pt_nuevo[k])]
    ci_nuevo.append(s)

#Una vez ordenados, calculamos el tiempo ocioso en cada máquina:

ct_aux=copy.deepcopy(ct_nuevo)
ci_aux=copy.deepcopy(ci_nuevo)
waiting_machines=[]
for i in range(0,instancia.machines):
    ct_aux[i].pop(-1)
    ci_aux[i].pop(0)
    aux=[(k-j) for k,j in zip(ci_aux[i],ct_aux[i])]
    waiting_machines.append(aux)

#Una vez tenemos todos los tiempos ociosos generados entre dos trabajos consecutivos en
```

```
    #cada máquina, realizamos la suma ponderada, teniendo en cuenta que cada máquina tiene
    #un peso distinto. El resultado de esta suma es el valor de nuestra función objetivo para
    #la secuencia dada

    suma_waiting_machines=[]
    funcion_objetivo=0
    sumapormaquina=0
    for i in range(0,len(waiting_machines)):
        for j in range(0,len(waiting_machines[i])):
            sumapormaquina=sumapormaquina+waiting_machines[i][j]
        suma_waiting_machines.append(sumapormaquina)
        funcion_objetivo=funcion_objetivo+suma_waiting_machines[i]*pesos_maquinas[i]
        sumapormaquina=0
    return funcion_objetivo

# Función para crear una secuencia aleatoria Jobshop.(También se podía haber hecho con
random_solution()) :

def secuencia_aleatoria_jobshop(instance):
    lista=[]
    for i in range(0,instance.jobs):
        for j in range(0,instance.machines):
            lista.append(i)
    random.shuffle(lista)
    return lista

# Función que calcula el fitness de una población:

def calcula_fitness_poblacion(instance,poblacion):
    fitness_poblacion=[]
    for i in range(0,len(poblacion)):
        fitness_poblacion.append(FUNCION_OBJETIVO(instance,poblacion[i]))
    return fitness_poblacion


# Función que crea la población inicial del algoritmo, para su funcionamiento,
# necesita la función anterior --> secuencia_aleatoria_jobshop() :

def creacion_poblacion_inicial(instance,tamaño_poblacion):
    poblacion=[]
    for i in range(0,tamaño_poblacion):
        poblacion.append(secuencia_aleatoria_jobshop(instance))
    return poblacion

# Función que crea el conjunto de secuencias individual extreme library:

def creacion_ind_extreme_lib(instance,poblacion):

    mejorseq_1=poblacion[random.randint(0,len(poblacion)-1)]
    mejorvalor_1=FUNCION_OBJETIVO(instancia,mejorseq_1)
    mejorseq_2=poblacion[random.randint(0,len(poblacion)-1)]
    while mejorseq_1==mejorseq_2:
        mejorseq_2=poblacion[random.randint(0,len(poblacion)-1)]
    mejorvalor_2=FUNCION_OBJETIVO(instancia,mejorseq_2)
    mejorseq_3=poblacion[random.randint(0,len(poblacion)-1)]
    while mejorseq_1==mejorseq_3 or mejorseq_2==mejorseq_3:
        mejorseq_3=poblacion[random.randint(0,len(poblacion)-1)]
```

```
    mejorvalor_3=FUNCION_OBJETIVO(instancia,mejorseq_3)

    ind_ext_lib=[]
    for i in range(0,len(poblacion)):
        mejorvalor_nuevo=FUNCION_OBJETIVO(instancia,poblacion[i])
        if(mejorvalor_1>mejorvalor_2 and mejorvalor_1>mejorvalor_3 and
mejorvalor_1>mejorvalor_nuevo):
            mejorvalor_1=mejorvalor_nuevo
            mejorseq_1=copy.deepcopy(poblacion[i])
        elif(mejorvalor_2>mejorvalor_1 and mejorvalor_2>mejorvalor_3 and
mejorvalor_2>mejorvalor_nuevo):
            mejorvalor_2=mejorvalor_nuevo
            mejorseq_2=copy.deepcopy(poblacion[i])
        elif(mejorvalor_3>mejorvalor_1 and mejorvalor_3>mejorvalor_2 and
mejorvalor_3>mejorvalor_nuevo):
            mejorvalor_3=mejorvalor_nuevo
            mejorseq_3=copy.deepcopy(poblacion[i])
        ind_ext_lib=[mejorseq_1,mejorseq_2,mejorseq_3]
    return ind_ext_lib

#Función que crea el conjunto de secuencias population extreme library:

def creacion_pop_extreme_lib(instance,poblacion):
    porcen_poblacion=int(20*len(poblacion)/100) #-> 20%
    lista_seq=[] #lista vacia de secuencias
    lista_fit=[] #lista vacia de fitness
    [(lista_fit.append(FUNCION_OBJETIVO(instance,x)),lista_seq.append(x)) for x in poblacion if
FUNCION_OBJETIVO(instance,x) not in lista_fit]
    #Despues de esta sentencia, se rellenan las listas anteriores,con secuencias únicas dentro de
poblacion
    pop_ext_lib=[]
    if (len(lista_seq)>=porcen_poblacion):
        for i in range(0,porcen_poblacion):
            pop_ext_lib.append(lista_seq[i])
    #Si no se alcanza el 20% de la poblacion, se añaden secuencias con fitness repetidos hasta
alcanzarlo
    elif(len(lista_seq)<porcen_poblacion):
        for i in range(0,len(lista_seq)):
            pop_ext_lib.append(lista_seq[i])
        for j in range(len(lista_seq),porcen_poblacion):
           pop_ext_lib.append(poblacion[j])
    return pop_ext_lib

#Funcion que calcula el cruce de dos secuencias mediante método POX:

def cruce_POX(instance,parent1,parent2):
    #Seleccion de subconjuntos jobset1 y jobset2 creado de forma random
    offspring1=[]
    offspring2=[]
    jobset1=[]
    jobset2=[]
    jobset1 = random.sample(range(instance.jobs),random.randint(1,instance.jobs-1))
    [jobset2.append(x) for x in range(0,instance.jobs) if x not in jobset1]

    #Creamos los descendientes : offspring1 y offspring2
    for i in range(0,len(parent1)):
        if parent1[i] in jobset2:
            offspring2.append(parent1[i])
```

```
        if parent2[i] in jobset2:
            offspring1.append(parent2[i])
    for i in range(0,len(parent1)):
        if parent1[i] in jobset1:
            offspring1.insert(i,parent1[i])
        if parent2[i] in jobset1:
            offspring2.insert(i,parent2[i])
    return offspring1,offspring2

#Función que lleva a cabo insertion dada dos posiciones:

def insertion(secuencia,posicion_1,posicion_2):
    aux=secuencia[posicion_1]
    secuencia.pop(posicion_1)
    secuencia.insert(posicion_2,aux)
    return secuencia

#Función que realiza mutación insertion.(Necesita la funcion anterior insertion):

def mutacion_insertion(instance,secuencia):
    posicion_1=random.randint(0,(instance.jobs*instance.machines)-1)
    posicion_2=random.randint(0,(instance.jobs*instance.machines)-1)
    while(posicion_1==posicion_2):
        posicion_1=random.randint(0,(instance.jobs*instance.machines)-1)
        posicion_2=random.randint(0,(instance.jobs*instance.machines)-1)
    secuencia=insertion(secuencia,posicion_1,posicion_2)
    return secuencia

#Función que compara dos secuencias y devuelve la que tiene mejor fitness:

def comparacion_secuencias(instance,mejorsecuencia,secuencia2):
      mejorvalor=FUNCION_OBJETIVO(instance,mejorsecuencia)
      valor_sec2=FUNCION_OBJETIVO(instance,secuencia2)
      if(valor_sec2<mejorvalor):
          mejorvalor=valor_sec2
          mejorsecuencia=copy.deepcopy(secuencia2)
      return mejorvalor,mejorsecuencia

#Función que compara dos secuencias y devuelve la de peor fitness (se usará para
# actualizar tanto ind_ext_lib como pop_ext_lib al final)

def comparacion_secuencias_peor(instance,peorsecuencia,secuencia2,i,indice_peor):
      peorvalor=FUNCION_OBJETIVO(instance,peorsecuencia)
      valor_sec2=FUNCION_OBJETIVO(instance,secuencia2)
      if(valor_sec2>peorvalor):
          peorvalor=valor_sec2
          peorsecuencia=copy.deepcopy(secuencia2)
          indice_peor=i
      return peorvalor,peorsecuencia,indice_peor


#Función que lleva a cabo la búsqueda global y actualiza la poblacion,
# creando una nueva poblacion

def BUSQUEDA_GLOBAL(instance,poblacion,ind_ext_lib,pop_ext_lib):
    parent1=[]
    parent2=[]
```

```
    parent3=[]
    nuevapoblacion=[]
    for i in range(0,len(poblacion)):
        #Current particle parent1
        parent1=copy.deepcopy(poblacion[i])

        #Cruce POX de parent1 con secuencia randomm de ind_ext_lib (parent2)
        #De aqui obtenemos dos descendientes --> offspring1 y offspring2

        valores_ind_ext=calcula_fitness_poblacion(instancia,ind_ext_lib)
        if valores_ind_ext[0]==valores_ind_ext[1] and valores_ind_ext[0]==valores_ind_ext[2]:
            parent2=ind_ext_lib[random.randint(0,len(ind_ext_lib)-1)]
        else:
            parent2=ind_ext_lib[random.randint(0,len(ind_ext_lib)-1)]
            while FUNCION_OBJETIVO(instancia,parent1)==FUNCION_OBJETIVO(instancia,parent2):
                parent2=ind_ext_lib[random.randint(0,len(ind_ext_lib)-1)]

        offspring1,offspring2=cruce_POX(instancia,parent1,parent2)

        #Cruce POX de parent1 con secuencia random de pop_ext_lib (parent3)
        #De aqui obtenemos dos descendientes --> offspring2 y offspring4
        b=0
        valores_pop_ext=calcula_fitness_poblacion(instancia,pop_ext_lib)
        valor_ref=valores_pop_ext[0]
        for k in range(1,len(pop_ext_lib)):
            valor_act=valores_pop_ext[k]
            if valor_ref!=valor_act:
                b=1
                break
        if b==0:
            parent3=pop_ext_lib[random.randint(0,len(pop_ext_lib)-1)]
        if b==1:
            parent3=pop_ext_lib[random.randint(0,len(pop_ext_lib)-1)]
            while  FUNCION_OBJETIVO(instancia,parent1)==FUNCION_OBJETIVO(instancia,parent3):
                parent3=pop_ext_lib[random.randint(0,len(pop_ext_lib)-1)]

        offspring3,offspring4=cruce_POX(instancia,parent1,parent3)

        #Mutacion insertion de parent1
        offspring5=[]
        offspring5=copy.deepcopy(poblacion[i])
        offspring5=mutacion_insertion(instancia,offspring5)

        #Comparamos los 5 descendientes y nos quedamos con el de mejor fitness
        mejorsecuencia=copy.deepcopy(offspring1)
        mejorvalor,mejorsecuencia=comparacion_secuencias(instancia,mejorsecuencia, offspring2)
        mejorvalor,mejorsecuencia=comparacion_secuencias(instancia,mejorsecuencia, offspring3)
        mejorvalor,mejorsecuencia=comparacion_secuencias(instancia,mejorsecuencia, offspring4)
        mejorvalor,mejorsecuencia=comparacion_secuencias(instancia,mejorsecuencia, offspring5)

        #Se va actualizando la poblacion
        nuevapoblacion.append(mejorsecuencia)
    return nuevapoblacion

# A continuación se definen funciones de generación de vecindades:

#Vecindario 1 --> N5 de Smutnicki (N5):
```

```
def N5(instancia,v1):
    #Esta función esta dividida en 3 apartados:
    #Apartado 1) Reordenación de los trabajos y codificación para el posterior cálculo del camino
crítico
    #Apartado 2) Cálculo del camino crítico y bloques críticos
    #Apartado 3) Decodificación y aplicación de N5 de Smutnicki
    #Apartado 4) Obtención del mejor vecino, el mejor valor de ese vecino y el vecindario N5

    ### APARTADO 1 ###

    #Reordenación de los trabajos. En este apartado realizamos una reordenación de forma que
    # iremos máquina a máquina asignando trabajos. Así, si tenemos 4 trabajos en la maquina 0,
    # El primer trabajo en la maquina 0 sera el 0, el segundo el 1 y asi sucesivamente.
    # Para la maquina 1, el primer trabajo será en el nuevo orden el trabajo 5 el segundo el 6 y
asi...

    #Obtencion del orden de los nuevos indices.Tras las siguientes operaciones se obtiene
    #una lista li_ind[] que contiene los trabajos en cada maquina en el orden que son procesados
    #Para ello tambien se ha ordenado sus ct y sus tiempos de proceso (ct_nuevo[] y pt_nuevo[]).
    #En este apartado también se ha calculado los tiempos de inicio de cada trabajo(ci_nuevo[])
    ct, job_order = instancia.ct(v1)
    tiempos=instancia.pt
    m=copy.deepcopy(ct)
    s=[]
    li=[]
    li_ct=[]
    li_ind=[]
    li_temp=[]
    for j in range(0,len(m)):
        for i in range(0,len(m[j])):
              li.append([m[j][i],i,tiempos[j][i]])
        li.sort()
        sort_ct=[]
        sort_index = []
        sort_temp=[]
        for x in li:
              sort_ct.append(x[0])
              sort_index.append(x[1])
              sort_temp.append(x[2])
        li_ct.append(sort_ct)
        li_ind.append(sort_index)
        li_temp.append(sort_temp)
        li=[]
    ct_nuevo=copy.deepcopy(li_ct)
    pt_nuevo=copy.deepcopy(li_temp)
    ci_nuevo=[]
    for k in range(0,instancia.machines):
        s=[(i-j) for i,j in zip(ct_nuevo[k],pt_nuevo[k])]
        ci_nuevo.append(s)


    ### APARTADO 2 ###

    #Cálculo del camino crítico. Todo lo que hemos hecho anteriormente es para preparar
    #el problema para el cálculo del camino crítico. Para ello se hace uso de la librería
    #criticalpath importada de forma pip. Con esta herramienta, introduciendo nodos y arcos y
    #somos capaces de obtener un camino crítico y posteriormente elaborar los bloques criticos
```

```
    #Paso 2.1) Cálculo de los nodos

    p = Node('project')
    cont=0
    listanodos=[]
    listanodos_amp=[]
    for i in range(0,instancia.machines):
        for j in range(0,instancia.jobs):
          if j==0:

listanodos.append(p.add(Node('{}'.format(cont),duration=pt_nuevo[i][j],lag=ci_nuevo[i][j])))
          else:

listanodos.append(p.add(Node('{}'.format(cont),duration=pt_nuevo[i][j],lag=ci_nuevo[i][j]-
ct_nuevo[i][j-1])))
          cont=cont+1
        listanodos_amp.append(listanodos)
        listanodos=[]

    #Paso 2.2) Cálculo de las dependencias (arcos)

    for i in range(0,instancia.machines):
        for j in range(0,instancia.jobs):

            for k in range(0,instancia.machines):
                if ct_nuevo[i][j] in ci_nuevo[k]:
                    indice= ci_nuevo[k].index(ct_nuevo[i][j])
                    p.link(listanodos_amp[i][j],listanodos_amp[k][indice])
    critical_path=p.get_critical_path()

    #Paso 2.3) Creación bloques críticos

    crit_path = [str(n) for n in p.get_critical_path()]
    bloques_criticos=[]
    b=[]
    i=0
    while i<len(crit_path)-1:
        while(int(crit_path[i])==int(crit_path[i+1])-1):
            if int(crit_path[i]) not in b:
                b.append(int(crit_path[i]))
            if int(crit_path[i+1]) not in b:
                b.append(int(crit_path[i+1]))
            i=i+1
            if i==len(crit_path)-1:
                break
        if(len(b)!=0):
              bloques_criticos.append(b)
        b=[]
        i=i+1

    ### APARTADO 3 ###

    #Aplicación de N5 de Smutnicki. Para ello decodificamos el resultado del apartado anterior.
    #Una vez localizados los intercambios a realizar en la secuencia aplicamos los intercambios
    #a como se indica en la generación de vecinos N5. El resultado son 5 vecinos adicionales

    #3.1) proceso de decodificacion:
```

```
    indice_global=0
    lista_dicc=[]
    for i in range(0,instancia.machines):
        for j in range(0,instancia.jobs):
            dicc={'orden': indice_global, 'trabajo':li_ind[i][j],'maquina': i,
'aparicion':instancia.rt[li_ind[i][j]].index(i)}
            lista_dicc.append(dicc)
            indice_global=indice_global+1

    #3.2) Obtención de vecinos mediante N5.
    #condición para obtener vecinos--> más de un bloque y cada bloque como mínimo dos operaciones

    vecino_1_n5=copy.deepcopy(v1)
    vecino_2_n5=copy.deepcopy(v1)
    vecino_3_n5=copy.deepcopy(v1)
    vecino_4_n5=copy.deepcopy(v1)
    vecino_5_n5=copy.deepcopy(v1)
    if len(bloques_criticos)>1:
            #intercambio de las dos ultimas pos del primer bloque. Se obtiene vecino_1_n5:
            sss=[i for i in range(len(v1)) if v1[i] == lista_dicc[bloques_criticos[0][-
1]]['trabajo']]
            posicion_secuencia_1=sss[lista_dicc[bloques_criticos[0][-1]]['aparicion']]
            sss=[i for i in range(len(v1)) if v1[i] == lista_dicc[bloques_criticos[0][-
2]]['trabajo']]
            posicion_secuencia_2=sss[lista_dicc[bloques_criticos[0][-2]]['aparicion']]
            aux=v1[posicion_secuencia_1]
            vecino_1_n5[posicion_secuencia_1]=v1[posicion_secuencia_2]
            vecino_1_n5[posicion_secuencia_2]=aux

            #intercambio de las dos primeras pos del ultimo bloque. Se obtiene vecino_2_n5:
            sss=[i for i in range(len(v1)) if v1[i] == lista_dicc[bloques_criticos[-
1][0]]['trabajo']]
            posicion_secuencia_1=sss[lista_dicc[bloques_criticos[-1][0]]['aparicion']]
            sss=[i for i in range(len(v1)) if v1[i] == lista_dicc[bloques_criticos[-
1][1]]['trabajo']]
            posicion_secuencia_2=sss[lista_dicc[bloques_criticos[-1][1]]['aparicion']]
            aux=v1[posicion_secuencia_1]
            vecino_2_n5[posicion_secuencia_1]=v1[posicion_secuencia_2]
            vecino_2_n5[posicion_secuencia_2]=aux

            #intercambio conjunto. Se obtiene vecino_3_n5
            sss=[i for i in range(len(v1)) if v1[i] == lista_dicc[bloques_criticos[0][-
1]]['trabajo']]
            posicion_secuencia_1=sss[lista_dicc[bloques_criticos[0][-1]]['aparicion']]
            sss=[i for i in range(len(v1)) if v1[i] == lista_dicc[bloques_criticos[0][-
2]]['trabajo']]
            posicion_secuencia_2=sss[lista_dicc[bloques_criticos[0][-2]]['aparicion']]
            aux=v1[posicion_secuencia_1]
            vecino_3_n5[posicion_secuencia_1]=v1[posicion_secuencia_2]
            vecino_3_n5[posicion_secuencia_2]=aux
            sss=[i for i in range(len(v1)) if v1[i] == lista_dicc[bloques_criticos[-
1][0]]['trabajo']]
            posicion_secuencia_1=sss[lista_dicc[bloques_criticos[-1][0]]['aparicion']]
            sss=[i for i in range(len(v1)) if v1[i] == lista_dicc[bloques_criticos[-
1][1]]['trabajo']]
            posicion_secuencia_2=sss[lista_dicc[bloques_criticos[-1][1]]['aparicion']]
```

```
            aux=v1[posicion_secuencia_1]
            vecino_3_n5[posicion_secuencia_1]=v1[posicion_secuencia_2]
            vecino_3_n5[posicion_secuencia_2]=aux

            #Intercambio en el/los bloque/s intermedio/s. Depende del tamaño del bloque,
            # se realizarán uno o dos intercambios. obteniendose así uno o dos vecinos
            #(vecino_4_n5 y vecino_5_n5):
            if len(bloques_criticos)>2:
                size_binter=len(bloques_criticos[1])
                indice_intermedio=1
                for i in range(1,len(bloques_criticos)-1):
                    if size_binter<len(bloques_criticos[i]):
                        size_binter=len(bloques_criticos[i])
                        indice_intermedio=i

                if len(bloques_criticos[indice_intermedio])==2:
                        sss=[i for i in range(len(v1)) if v1[i] ==
lista_dicc[bloques_criticos[indice_intermedio][-1]]['trabajo']]
                        posicion_secuencia_1=sss[lista_dicc[bloques_criticos[indice_intermedio][-
1]]['aparicion']]
                        sss=[i for i in range(len(v1)) if v1[i] ==
lista_dicc[bloques_criticos[indice_intermedio][-2]]['trabajo']]
                        posicion_secuencia_2=sss[lista_dicc[bloques_criticos[indice_intermedio][-
2]]['aparicion']]
                        aux=v1[posicion_secuencia_1]
                        vecino_4_n5[posicion_secuencia_1]=v1[posicion_secuencia_2]
                        vecino_4_n5[posicion_secuencia_2]=aux

                elif len(bloques_criticos[indice_intermedio])>2:
                       sss=[i for i in range(len(v1)) if v1[i] ==
lista_dicc[bloques_criticos[indice_intermedio][-1]]['trabajo']]
                      posicion_secuencia_1=sss[lista_dicc[bloques_criticos[indice_intermedio][-
1]]['aparicion']]
                       sss=[i for i in range(len(v1)) if v1[i] ==
lista_dicc[bloques_criticos[indice_intermedio][-2]]['trabajo']]
                       posicion_secuencia_2=sss[lista_dicc[bloques_criticos[indice_intermedio][-
2]]['aparicion']]
                       aux=v1[posicion_secuencia_1]
                       vecino_4_n5[posicion_secuencia_1]=v1[posicion_secuencia_2]
                       vecino_4_n5[posicion_secuencia_2]=aux

                       sss=[i for i in range(len(v1)) if v1[i] ==
lista_dicc[bloques_criticos[indice_intermedio][0]]['trabajo']]

posicion_secuencia_1=sss[lista_dicc[bloques_criticos[indice_intermedio][0]]['aparicion']]
                       sss=[i for i in range(len(v1)) if v1[i] ==
lista_dicc[bloques_criticos[indice_intermedio][1]]['trabajo']]

posicion_secuencia_2=sss[lista_dicc[bloques_criticos[indice_intermedio][1]]['aparicion']]
                       aux=v1[posicion_secuencia_1]
                       vecino_5_n5[posicion_secuencia_1]=v1[posicion_secuencia_2]
                       vecino_5_n5[posicion_secuencia_2]=aux
    secuencias_nuevas=[]
    secuencias_nuevas.append(vecino_1_n5)
    secuencias_nuevas.append(vecino_2_n5)
    secuencias_nuevas.append(vecino_3_n5)
    secuencias_nuevas.append(vecino_4_n5)
    secuencias_nuevas.append(vecino_5_n5)
```

```
    ### APARTADO 4 ###

    #cálculo de la mejor secuencia dentro del vecindario y devolución mediante return de
    # mejor fitness - mejor vecino - vecindario n5
    mejorsecuencia=copy.deepcopy(secuencias_nuevas[0])
    for i in range(1,len(secuencias_nuevas)):

mejorvalor,mejorsecuencia=comparacion_secuencias(instancia,mejorsecuencia,secuencias_nuevas[i])
    return secuencias_nuevas,mejorsecuencia,mejorvalor


#Vecindario 2 --> Método Random Whole Neighborhood (RWN):

def RWN(instance,secuencia):
    if (len(poblacion)<200):
        parametro=4
        n=[]
        secuencia_nueva=[]
        secuencias_nuevas=[]
        numperm=0
        s=random.sample(range(0,len(poblacion[0])),parametro)
        perm=permutations(s)
        for i in list(perm):
            n.append(i)
            numperm=numperm+1
        for i in range(0,numperm):
            secuencia_nueva=copy.deepcopy(secuencia)
            secuencia_nueva[n[i][0]]=secuencia[n[0][0]]
            secuencia_nueva[n[i][1]]=secuencia[n[0][1]]
            secuencia_nueva[n[i][2]]=secuencia[n[0][2]]
            secuencia_nueva[n[i][3]]=secuencia[n[0][3]]
            secuencias_nuevas.append(secuencia_nueva)
    else:
        parametro=5
        n=[]
        secuencia_nueva=[]
        secuencias_nuevas=[]
        numperm=0
        s=random.sample(range(0,len(poblacion[0])),parametro)
        perm=permutations(s)
        for i in list(perm):
            n.append(i)
            numperm=numperm+1
        for i in range(0,numperm):
            secuencia_nueva=copy.deepcopy(secuencia)
            secuencia_nueva[n[i][0]]=secuencia[n[0][0]]
            secuencia_nueva[n[i][1]]=secuencia[n[0][1]]
            secuencia_nueva[n[i][2]]=secuencia[n[0][2]]
            secuencia_nueva[n[i][3]]=secuencia[n[0][3]]
            secuencia_nueva[n[i][4]]=secuencia[n[0][4]]
            secuencias_nuevas.append(secuencia_nueva)

    # calculo de la mejor secuencia dentro del vecindario
    mejorsecuencia=copy.deepcopy(secuencias_nuevas[0])
    for i in range(1,len(secuencias_nuevas)):
```

```
mejorvalor,mejorsecuencia=comparacion_secuencias(instance,mejorsecuencia,secuencias_nuevas[i])
    return secuencias_nuevas,mejorsecuencia,mejorvalor

#Vecindario 3 --> Random Reverse Neighborhood (RRN):

def RRN(instance,secuencia):
    secuencia_nueva=[]
    secuencias_nuevas=[]
    for i in range(0,20):
        s=random.sample(range(0,len(poblacion[0])),2)
        secuencia_nueva=copy.deepcopy(secuencia)
        aux=secuencia_nueva[s[0]]
        secuencia_nueva[s[0]]=secuencia_nueva[s[1]]
        secuencia_nueva[s[1]]=aux
        secuencias_nuevas.append(secuencia_nueva)

    # calculo de la mejor secuencia dentro del vecindario
    #mejorsecuencia=secuencias_nuevas[0].copy()
    mejorsecuencia=copy.deepcopy(secuencias_nuevas[0])
    for i in range(1,len(secuencias_nuevas)):

mejorvalor,mejorsecuencia=comparacion_secuencias(instance,mejorsecuencia,secuencias_nuevas[i])
    return secuencias_nuevas,mejorsecuencia,mejorvalor


#Vecindario 4 --> Two Point Exchange Neighborhood (TEN):

def TEN(instance,secuencia):
    s=random.sample(range(0,len(poblacion[0])),2)
    secuencia_nueva=copy.deepcopy(secuencia)
    aux=secuencia_nueva[s[0]]
    secuencia_nueva[s[0]]=secuencia_nueva[s[1]]
    secuencia_nueva[s[1]]=aux

    # calculo de la mejor secuencia dentro del vecindario
    mejorsecuencia=copy.deepcopy(secuencia)
    mejorvalor,mejorsecuencia=comparacion_secuencias(instance,mejorsecuencia,secuencia_nueva)
    return secuencia_nueva,mejorsecuencia,mejorvalor

#Vecindario 5 --> Two Random Insert Neighborhood (RIN):

def RIN(instance,secuencia):
    s=random.sample(range(0,len(poblacion[0])),2)
    secuencia_nueva=copy.deepcopy(secuencia)
    aux=secuencia_nueva.pop(s[0])
    secuencia_nueva.insert(s[1],aux)

    # calculo de la mejor secuencia dentro del vecindario
    mejorsecuencia=copy.deepcopy(secuencia)
    mejorvalor,mejorsecuencia=comparacion_secuencias(instance,mejorsecuencia,secuencia_nueva)
    return secuencia_nueva,mejorsecuencia,mejorvalor

#Función de búsqueda local. Haciendo uso de las funciones anteriores de creación
#de vecindarios, devuelve una nueva población con las mejores secuencias de busqueda local.
#Esta función también devuelve la mejor secuencia de este conjunto.

def BUSQUEDA_LOCAL(instance,poblacion):
```

```
    mejorsec=[]
    vecindario1=[]
    vecindario2=[]
    vecindario3=[]
    vecino4=[]
    vecino5=[]
    mejores_secuencias_VNS=[]
    for i in range(0,len(poblacion)):
        b=0
        mejorsec_0=copy.deepcopy(poblacion[i])
        mejorval_0=FUNCION_OBJETIVO(instance,mejorsec_0)
        while b==0:
              vecindario1,mejorsec,mejorval=N5(instance,mejorsec_0)
              if(mejorval<mejorval_0):
                 mejorval_0=mejorval
                 mejorsec_0=copy.deepcopy(mejorsec)
                 b=0
              else:
                  vecindario2,mejorsec,mejorval=RWN(instance,mejorsec_0)
                  if(mejorval<mejorval_0):
                      mejorval_0=mejorval
                      mejorsec_0=copy.deepcopy(mejorsec)
                      b=0
                  else:
                      vecindario3,mejorsec,mejorval=RRN(instance,mejorsec_0)
                      if(mejorval<mejorval_0):
                          mejorval_0=mejorval
                          mejorsec_0=copy.deepcopy(mejorsec)
                          b=0
                      else:
                          vecino4,mejorsec,mejorval=TEN(instance,mejorsec_0)
                          if(mejorval<mejorval_0):
                              mejorval_0=mejorval
                              mejorsec_0=copy.deepcopy(mejorsec)
                              b=0
                          else:
                              vecino5,mejorsec,mejorval=RIN(instance,mejorsec_0)
                              if(mejorval<mejorval_0):
                                  mejorval_0=mejorval
                                  mejorsec_0=copy.deepcopy(mejorsec)
                                  b=0
                              else:
                                  mejores_secuencias_VNS.append(mejorsec_0)
                                  b=1

    # calculo de la mejor secuencia dentro del vecindario
    mejorsecuencia=copy.deepcopy(mejores_secuencias_VNS[0])
    for i in range(1,len(mejores_secuencias_VNS)):

mejorvalor,mejorsecuencia=comparacion_secuencias(instance,mejorsecuencia,mejores_secuencias_VNS[i
])
    return mejores_secuencias_VNS,mejorsecuencia,mejorvalor

#Función que actualiza Individual Extreme Library (ind_ext_lib)
# y Population Extreme Library (pop_ext_lib)
```

```
def actualizacion_libs(instance,poblacion,ind_ext_lib,pop_ext_lib):
    for k in range(0,len(poblacion)):
        peorsecuencia=copy.deepcopy(ind_ext_lib[0])
        indice_peor=0
        for i in range(1,len(ind_ext_lib)):

peorvalor,peorsecuencia,indice_peor=comparacion_secuencias_peor(instance,peorsecuencia,ind_ext_li
b[i],i,indice_peor)
        mejorsecuencia=copy.deepcopy(poblacion[k])
        mejorvalor,mejorsecuencia=comparacion_secuencias(instance,mejorsecuencia,peorsecuencia)
        fitness_ind=calcula_fitness_poblacion(instance,ind_ext_lib)
        if mejorsecuencia not in ind_ext_lib:
            if mejorvalor not in fitness_ind:
                ind_ext_lib.pop(indice_peor)
                ind_ext_lib.insert(indice_peor,mejorsecuencia)

        peorsecuencia=copy.deepcopy(pop_ext_lib[0])
        indice_peor=0
        for j in range(1,len(pop_ext_lib)):

peorvalor,peorsecuencia,indice_peor=comparacion_secuencias_peor(instance,peorsecuencia,pop_ext_li
b[j],j,indice_peor)
        mejorsecuencia=copy.deepcopy(poblacion[k])
        mejorvalor,mejorsecuencia=comparacion_secuencias(instance,mejorsecuencia,peorsecuencia)
        fitness_pop=calcula_fitness_poblacion(instance,pop_ext_lib)
        if mejorsecuencia not in pop_ext_lib:
            if mejorvalor not in fitness_pop:
                pop_ext_lib.pop(indice_peor)
                pop_ext_lib.insert(indice_peor,mejorsecuencia)

    return ind_ext_lib,pop_ext_lib


#-------------------------------------------------------------------------------
''' 3) Este apartado equivale al "main", ejecución del algoritmo HPV (Hybrid PSO VNS)'''

### ---------------------- MAIN - EJECUCIÓN DE ALGORITMO  ------------------- ###

#         Se establece para ejecutar Las instancias FT10,FT20 y las 30 instancias desde LA1-LA30

#          Consideraciones a tener en cuenta:
#       a) La instancia va a ser ejecutada 10 veces.
#       b) Se han fijado las variables Número de población y tiempo de parada a:
#           - Numero de población de 10 secuencias  --> tamaño_poblacion=100
#           - Tiempo de parada a los 15 segundos --> while(tiempo_ejecucion<15).
#       c) El archivo txt de la instancia ha de estar en el mismo directorio.
#       d) como resultado el programa genera un fichero txt. con los resultados.

for i in range(2):

    archivo='test_jobshop_ft'+ str(i+1)+'pru.txt'
    lista_exp=[]
    lista_exp_2=[]


    for exp in range(0,10):

        #IMPORTANTE! Antes de continuar, dentro de la función def FUNCION_OBJETIVO ,
```

```
        #hay que definir los pesos de cada máquina

        #Inicio de tiempo de ejecución:
        start=time.time()

        #Carga de la instancia Jobshop:
        instancia = JobShop(archivo)

        #Definición del tamaño de la población (population size):
        tamaño_poblacion=10

        #Creación de una población inicial con secuencias random:
        poblacion=creacion_poblacion_inicial(instancia,tamaño_poblacion)

        #Creación de individual extreme library:
        ind_ext_lib=creacion_ind_extreme_lib(instancia,poblacion)

        #Creación de la population extreme library:
        pop_ext_lib=creacion_pop_extreme_lib(instancia,poblacion)

        end=time.time()
        tiempo_ejecucion=end-start

        while(tiempo_ejecucion<15):

            #Búsqueda global (PSO) --> PASO 3
            nuevapoblacion=BUSQUEDA_GLOBAL(instancia, poblacion,ind_ext_lib,pop_ext_lib)

            #Búsqueda local (VNS) --> PASO 4

mejores_secuencias_VNS,bestseq_VNS,bestfitness_VNS=BUSQUEDA_LOCAL(instancia,nuevapoblacion)

            #Actualización de población
            poblacion=copy.deepcopy(mejores_secuencias_VNS)

            #actualizacion de librerias pop_ext e ind_ext --> PASO 5

ind_ext_lib,pop_ext_lib=actualizacion_libs(instancia,poblacion,ind_ext_lib,pop_ext_lib)

            end=time.time()
            tiempo_ejecucion=end-start

        # calculo de la mejor secuencia y el mejor valor de ind_ext_lib al final del algoritmo
        bestsecuence=copy.deepcopy(ind_ext_lib[0])
        for i in range(1,len(ind_ext_lib)):
             bestvalue,bestsecuence=comparacion_secuencias(instancia,bestsecuence,ind_ext_lib[i])

        lista_exp.append(bestvalue)
        lista_exp_2.append(instancia.Cmax(bestsecuence))

    with open('resultados_HPV_FO_t15_p10.txt','a') as archivo_resultados:
        archivo_resultados.write(archivo + ' ' + 'F.O' + str(lista_exp) + ' ' + '\n')
        archivo_resultados.write(archivo + ' ' + 'MAKESPAN' + str(lista_exp_2) + ' ' + '\n')
        archivo_resultados.write('\n')

for i in range(30):
```

```
    archivo='test_jobshop_LA'+ str(i+1)+'.txt'
    lista_exp=[]
    lista_exp_2=[]

    for exp in range(0,10):

        #IMPORTANTE! Antes de continuar, dentro de la función def FUNCION_OBJETIVO ,
        #hay que definir los pesos de cada máquina

        #Inicio de tiempo de ejecución:
        start=time.time()

        instancia = JobShop(archivo)

        #Definición del tamaño de la población (population size):
        tamaño_poblacion=10

        #Creación de una población inicial con secuencias random:
        poblacion=creacion_poblacion_inicial(instancia,tamaño_poblacion)

        #Creación de individual extreme library:
        ind_ext_lib=creacion_ind_extreme_lib(instancia,poblacion)

        #Creación de la population extreme library:
        pop_ext_lib=creacion_pop_extreme_lib(instancia,poblacion)

        end=time.time()
        tiempo_ejecucion=end-start

        while(tiempo_ejecucion<15):

            #Búsqueda global (PSO) --> PASO 3
            nuevapoblacion=BUSQUEDA_GLOBAL(instancia, poblacion,ind_ext_lib,pop_ext_lib)

            #Búsqueda local (VNS) --> PASO 4

mejores_secuencias_VNS,bestseq_VNS,bestfitness_VNS=BUSQUEDA_LOCAL(instancia,nuevapoblacion)

            #Actualización de población
            poblacion=copy.deepcopy(mejores_secuencias_VNS)

            #actualizacion de librerias pop_ext e ind_ext --> PASO 5

ind_ext_lib,pop_ext_lib=actualizacion_libs(instancia,poblacion,ind_ext_lib,pop_ext_lib)

            end=time.time()
            tiempo_ejecucion=end-start

        # calculo de la mejor secuencia y el mejor valor de ind_ext_lib al final del algoritmo
        bestsecuence=copy.deepcopy(ind_ext_lib[0])
        for i in range(1,len(ind_ext_lib)):
             bestvalue,bestsecuence=comparacion_secuencias(instancia,bestsecuence,ind_ext_lib[i])

        lista_exp.append(bestvalue)
        lista_exp_2.append(instancia.Cmax(bestsecuence))

    with open('resultados_HPV_FO_t15_p10.txt','a') as archivo_resultados:
        archivo_resultados.write(archivo + ' ' + 'F.O' + str(lista_exp) + ' ' + '\n')
```

```
        archivo_resultados.write(archivo + ' ' + 'MAKESPAN' + str(lista_exp_2) + ' ' + '\n')
        archivo_resultados.write('\n')


#-------------------------------------------------------------------------------
''' 4) Representación gráfica de la mejor solución en un diagrama de Gantt '''

### --------------------- REPRESENTACIÓN DIAGRAMA DE GANTT  ------------------- ###
"""
instancia.print_schedule(bestsecuence)
"""
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

A.2 Código SA

```
#-------------------------------------------------------------------------------

''' 1) En primer lugar se definen las librerías '''

### ---------------------DEFINICIÓN DE LIBRERIAS/MÓDULOS------------------- ###

from scheptk.scheptk import *
from scheptk.util import *
import random
from itertools import permutations
import copy
import time
import math

#-------------------------------------------------------------------------------

''' 2) En este apartado se definen todas las funciones'''

###------------------------- DEFINICIÓN DE FUNCIONES----------------------- ###

#Función que devuelve el valor de la función objetivo
# F.O --> MINIMIZAR TIEMPO OCIOSO EN LAS MAQUINAS (PONDERADO)

def FUNCION_OBJETIVO(instancia,secuencia):

    #Funcion objetivo - MIN CORE IDLE TIME PONDERADO (máquinas con distinto peso).
    #Como las máquinas tienen distinto peso, en primer lugar vamos a asignar los pesos a las
    #máquinas en el vector pesos_maquinas=[]:
    #Si no se quiere tener en cuenta la ponderación, poner todos los pesos=1
    ######################

    pesos_maquinas=[1,1,1,1,1,1,1,1,1,1]

    ######################

    #Obtencion del orden de los nuevos indices.Tras las siguientes operaciones se obtiene
    #una lista li_ind[] que contiene los trabajos en cada maquina en el orden que son procesados
    #Para ello tambien se ha ordenado sus ct y sus tiempos de proceso (ct_nuevo[] y pt_nuevo[]).
```

```
#En este apartado también se ha calculado los tiempos de inicio de cada trabajo(ci_nuevo[])

ct, job_order = instancia.ct(secuencia)
tiempos=instancia.pt
m=copy.deepcopy(ct)
s=[]
li=[]
li_ct=[]
li_ind=[]
li_temp=[]
for j in range(0,len(m)):
    for i in range(0,len(m[j])):
          li.append([m[j][i],i,tiempos[j][i]])
    li.sort()
    sort_ct=[]
    sort_index = []
    sort_temp=[]
    for x in li:
          sort_ct.append(x[0])
          sort_index.append(x[1])
          sort_temp.append(x[2])
    li_ct.append(sort_ct)
    li_ind.append(sort_index)
    li_temp.append(sort_temp)
    li=[]
ct_nuevo=copy.deepcopy(li_ct)
pt_nuevo=copy.deepcopy(li_temp)
ci_nuevo=[]
for k in range(0,instancia.machines):
    s=[(i-j) for i,j in zip(ct_nuevo[k],pt_nuevo[k])]
    ci_nuevo.append(s)

#Una vez ordenados, calculamos el tiempo ocioso en cada máquina:

ct_aux=copy.deepcopy(ct_nuevo)
ci_aux=copy.deepcopy(ci_nuevo)
waiting_machines=[]
for i in range(0,instancia.machines):
    ct_aux[i].pop(-1)
    ci_aux[i].pop(0)
    aux=[(k-j) for k,j in zip(ci_aux[i],ct_aux[i])]
    waiting_machines.append(aux)

#Una vez tenemos todos los tiempos ociosos generados entre dos trabajos consecutivos en
#cada máquina, realizamos la suma ponderada, teniendo en cuenta que cada máquina tiene
#un peso distinto. El resultado de esta suma es el valor de nuestra función objetivo para
#la secuencia dada

suma_waiting_machines=[]
funcion_objetivo=0
sumapormaquina=0
for i in range(0,len(waiting_machines)):
    for j in range(0,len(waiting_machines[i])):
        sumapormaquina=sumapormaquina+waiting_machines[i][j]
    suma_waiting_machines.append(sumapormaquina)
    funcion_objetivo=funcion_objetivo+suma_waiting_machines[i]*pesos_maquinas[i]
    sumapormaquina=0
return funcion_objetivo
```

```
# Función para crear una secuencia aleatoria Jobshop.(También se podía haber hecho con
random_solution()) :

def secuencia_aleatoria_jobshop(instance):
    lista=[]
    for i in range(0,instance.jobs):
        for j in range(0,instance.machines):
            lista.append(i)
    random.shuffle(lista)
    return lista

#Función que calcula el fitness de una población:

def calcula_fitness_poblacion(instance,poblacion):
    fitness_poblacion=[]
    for i in range(0,len(poblacion)):
        fitness_poblacion.append(FUNCION_OBJETIVO(instance,poblacion[i]))
    return fitness_poblacion

# Función que crea la población inicial del algoritmo, para su funcionamiento,
# necesita la función anterior --> secuencia_aleatoria_jobshop() :

def creacion_poblacion_inicial(instance,tamaño_poblacion):
    poblacion=[]
    for i in range(0,tamaño_poblacion):
        poblacion.append(secuencia_aleatoria_jobshop(instance))
    return poblacion

#Funcion que calcula el cruce de dos secuencias mediante método POX:

def cruce_POX(instance,parent1,parent2):
    #Seleccion de subconjuntos jobset1 y jobset2 creado de forma random
    offspring1=[]
    offspring2=[]
    jobset1=[]
    jobset2=[]
    jobset1 = random.sample(range(instance.jobs),random.randint(1,instance.jobs-1))
    [jobset2.append(x) for x in range(0,instance.jobs) if x not in jobset1]

    #Creamos los descendientes : offspring1 y offspring2
    for i in range(0,len(parent1)):
        if parent1[i] in jobset2:
            offspring2.append(parent1[i])
        if parent2[i] in jobset2:
            offspring1.append(parent2[i])
    for i in range(0,len(parent1)):
        if parent1[i] in jobset1:
            offspring1.insert(i,parent1[i])
        if parent2[i] in jobset1:
            offspring2.insert(i,parent2[i])
    return offspring1,offspring2

#Función que intercambia dos posiciones random de una secuencia:

def intercambio_random(instance,secuencia):
```

```
    s=random.sample(range(0,len(secuencia)),2)
    secuencia_nueva=copy.deepcopy(secuencia)
    aux=secuencia_nueva[s[0]]
    secuencia_nueva[s[0]]=secuencia_nueva[s[1]]
    secuencia_nueva[s[1]]=aux
    return secuencia_nueva


def intercambio(instance,secuencia,posicion1,posicion2):
    s=[]
    s.append(posicion1)
    s.append(posicion2)
    secuencia_nueva=copy.deepcopy(secuencia)
    aux=secuencia_nueva[s[0]]
    secuencia_nueva[s[0]]=secuencia_nueva[s[1]]
    secuencia_nueva[s[1]]=aux
    return secuencia_nueva

#Función que lleva a cabo insertion dada dos posiciones:

def insertion(secuencia,posicion_1,posicion_2):
    aux=secuencia[posicion_1]
    secuencia.pop(posicion_1)
    secuencia.insert(posicion_2,aux)
    return secuencia

#Función que compara dos secuencias y devuelve la que tiene mejor fitness:

def comparacion_secuencias(instance,mejorsecuencia,secuencia2):
      mejorvalor=FUNCION_OBJETIVO(instance,mejorsecuencia)
      valor_sec2=FUNCION_OBJETIVO(instance,secuencia2)
      if(valor_sec2<mejorvalor):
          mejorvalor=valor_sec2
          mejorsecuencia=copy.deepcopy(secuencia2)
      return mejorvalor,mejorsecuencia

#función que devuelve la mejor secuencia de una población:

def mejorsec_poblacion(instance,poblacion):
    #mejorsecuencia=[]
    mejorsecuencia=copy.deepcopy(poblacion[0])
    for i in range(1,len(poblacion)):
        mejorvalor,mejorsecuencia=comparacion_secuencias(instance,mejorsecuencia,poblacion[i])
    return mejorsecuencia,mejorvalor

#Vecindad para evaluación local: Random Reverse Neighborhood (RRN).
#Con el primer bucle for podemos indicar el número de vecinos por intercambio de
#dos posiciones que va a contener el vecindario.

def RRN(instance,secuencia):
    secuencia_nueva=[]
    secuencias_nuevas=[]
    for i in range(0,100):
        s=random.sample(range(0,len(secuencia)),2)
        secuencia_nueva=copy.deepcopy(secuencia)
        aux=secuencia_nueva[s[0]]
        secuencia_nueva[s[0]]=secuencia_nueva[s[1]]
        secuencia_nueva[s[1]]=aux
```

```
        secuencias_nuevas.append(secuencia_nueva)

    # calculo de la mejor secuencia dentro del vecindario
    mejorsecuencia=copy.deepcopy(secuencias_nuevas[0])
    for i in range(1,len(secuencias_nuevas)):

mejorvalor,mejorsecuencia=comparacion_secuencias(instance,mejorsecuencia,secuencias_nuevas[i])
    return secuencias_nuevas,mejorsecuencia,mejorvalor

#Función que crea la vecindad Adjacent-Swap:

def ADJACENT_SWAP(instance,secuencia):
    seq_mejor=copy.deepcopy(secuencia)
    #fit_mejor=instancia.Cmax(secuencia)
    fit_mejor=FUNCION_OBJETIVO(instance,secuencia)
    posicion_1=0
    posicion_2=1
    for i in range(0,len(secuencia)-1):
        vecino=intercambio(instance,secuencia,posicion_1,posicion_2)
        if vecino!=seq_mejor:
            #fit_vecino=instancia.Cmax(vecino)
            fit_vecino=FUNCION_OBJETIVO(instance,vecino)
            if (fit_vecino<fit_mejor):
                fit_mejor=fit_vecino
                seq_mejor=copy.deepcopy(vecino)
        print(vecino)
        posicion_1=posicion_1+1
        posicion_2=posicion_2+1
    return seq_mejor,fit_mejor


#-------------------------------------------------------------------------------
''' 3) Este apartado equivale al "main", ejecución Simulated Annealing'''

### -------------------- MAIN - EJECUCIÓN DE ALGORITMO  ------------------- ###

for i in range(2):

    archivo='test_jobshop_ft'+ str(i+1)+'pru.txt'
    lista_exp=[]
    lista_exp_2=[]

    for exp in range(0,10):

        #Inicio de tiempo de ejecución:
        start=time.time()

        #Carga de la instancia Jobshop:
        instancia = JobShop(archivo)

        #Se crea una solución inicial:
        mejorsecuencia=[]
        mejorsecuencia=instancia.random_solution()
        fit_mejorvalor=FUNCION_OBJETIVO(instancia,mejorsecuencia)

        #Selección de Temperatura inicial:
        T=1
```

```
        #Selección del paso r:
        r=0.9

        end=time.time()
        tiempo_ejecucion=end-start

        while(tiempo_ejecucion<15):
            vecindario,mejorsec_vec,mejorfit_vec=RRN(instancia,mejorsecuencia)
            if(mejorfit_vec<fit_mejorvalor):
                fit_mejorvalor=mejorfit_vec
                mejorsecuencia=copy.deepcopy(mejorsec_vec)
            else:
                #Creación de distancia entre soluciones D:
                D= mejorfit_vec-fit_mejorvalor
                random_num=random.randint(0,1000000)
                random_num=random_num/1000000
                if(random_num<math.exp(-D/T)):
                    fit_mejorvalor=mejorfit_vec
                    mejorsecuencia=copy.deepcopy(mejorsec_vec)
            T=r*T
            end=time.time()
            tiempo_ejecucion=end-start

        lista_exp.append(fit_mejorvalor)
        lista_exp_2.append(instancia.Cmax(mejorsecuencia))

    with open('resultados_SA_FO_t15.txt','a') as archivo_resultados:
        archivo_resultados.write(archivo + ' ' + 'F.O' + str(lista_exp) + ' ' + '\n')
        archivo_resultados.write(archivo + ' ' + 'MAKESPAN' + str(lista_exp_2) + ' ' + '\n')
        archivo_resultados.write('\n')


for i in range(30):

    archivo='test_jobshop_LA'+ str(i+1)+'.txt'
    lista_exp=[]
    lista_exp_2=[]

    for exp in range(0,10):

        #Inicio de tiempo de ejecución:
        start=time.time()

        #Carga de la instancia Jobshop:
        instancia = JobShop(archivo)

        #Se crea una solución inicial:
        mejorsecuencia=[]
        mejorsecuencia=instancia.random_solution()
        fit_mejorvalor=FUNCION_OBJETIVO(instancia,mejorsecuencia)

        #Selección de Temperatura inicial:
        T=1

        #Selección del paso r:
        r=0.9
```

```
        end=time.time()
        tiempo_ejecucion=end-start

        while(tiempo_ejecucion<15):
            vecindario,mejorsec_vec,mejorfit_vec=RRN(instancia,mejorsecuencia)
            if(mejorfit_vec<fit_mejorvalor):
                fit_mejorvalor=mejorfit_vec
                mejorsecuencia=copy.deepcopy(mejorsec_vec)
            else:
                #Creación de distancia entre soluciones D:
                D= mejorfit_vec-fit_mejorvalor
                random_num=random.randint(0,1000000)
                random_num=random_num/1000000
                if(random_num<math.exp(-D/T)):
                    fit_mejorvalor=mejorfit_vec
                    mejorsecuencia=copy.deepcopy(mejorsec_vec)
            T=r*T
            end=time.time()
            tiempo_ejecucion=end-start

        lista_exp.append(fit_mejorvalor)
        lista_exp_2.append(instancia.Cmax(mejorsecuencia))

    with open('resultados_SA_FO_t15.txt','a') as archivo_resultados:
        archivo_resultados.write(archivo + ' ' + 'F.O' + str(lista_exp) + ' ' + '\n')
        archivo_resultados.write(archivo + ' ' + 'MAKESPAN' + str(lista_exp_2) + ' ' + '\n')
        archivo_resultados.write('\n')
```

A.3 Código AG

```
#-------------------------------------------------------------------------------
-

''' 1) En primer lugar se definen las librerias'''

### ---------------------DEFINICIÓN DE LIBRERIAS/MÓDULOS------------------- ###

from scheptk.scheptk import *
from scheptk.util import *
import random
from itertools import permutations
import copy
import time

#-------------------------------------------------------------------------------

''' 2) En este apartado se definen todas las funciones'''

###------------------------ DEFINICIÓN DE FUNCIONES----------------------- ###

#Función que devuelve el valor de la función objetivo
# F.O --> MINIMIZAR TIEMPO OCIOSO EN LAS MAQUINAS (PONDERADO)
```

```
def FUNCION_OBJETIVO(instancia,secuencia):

    #Funcion objetivo - MIN CORE IDLE TIME PONDERADO (máquinas con distinto peso).
    #Como las máquinas tienen distinto peso, en primer lugar vamos a asignar los pesos a las
    #máquinas en el vector pesos_maquinas=[]:
    #Si no se quiere tener en cuenta la ponderación, poner todos los pesos=1
    ######################

    pesos_maquinas=[1,1,1,1,1,1,1,1,1,1]

    ######################

    #Obtencion del orden de los nuevos indices.Tras las siguientes operaciones se obtiene
    #una lista li_ind[] que contiene los trabajos en cada maquina en el orden que son procesados
    #Para ello tambien se ha ordenado sus ct y sus tiempos de proceso (ct_nuevo[] y pt_nuevo[]).
    #En este apartado también se ha calculado los tiempos de inicio de cada trabajo(ci_nuevo[])

    ct, job_order = instancia.ct(secuencia)
    tiempos=instancia.pt
    m=copy.deepcopy(ct)
    s=[]
    li=[]
    li_ct=[]
    li_ind=[]
    li_temp=[]
    for j in range(0,len(m)):
        for i in range(0,len(m[j])):
              li.append([m[j][i],i,tiempos[j][i]])
        li.sort()
        sort_ct=[]
        sort_index = []
        sort_temp=[]
        for x in li:
              sort_ct.append(x[0])
              sort_index.append(x[1])
              sort_temp.append(x[2])
        li_ct.append(sort_ct)
        li_ind.append(sort_index)
        li_temp.append(sort_temp)
        li=[]
    ct_nuevo=copy.deepcopy(li_ct)
    pt_nuevo=copy.deepcopy(li_temp)
    ci_nuevo=[]
    for k in range(0,instancia.machines):
        s=[(i-j) for i,j in zip(ct_nuevo[k],pt_nuevo[k])]
        ci_nuevo.append(s)

    #Una vez ordenados, calculamos el tiempo ocioso en cada máquina:

    ct_aux=copy.deepcopy(ct_nuevo)
    ci_aux=copy.deepcopy(ci_nuevo)
    waiting_machines=[]
    for i in range(0,instancia.machines):
        ct_aux[i].pop(-1)
        ci_aux[i].pop(0)
        aux=[(k-j) for k,j in zip(ci_aux[i],ct_aux[i])]
        waiting_machines.append(aux)
```

```
    #Una vez tenemos todos los tiempos ociosos generados entre dos trabajos consecutivos en
    #cada máquina, realizamos la suma ponderada, teniendo en cuenta que cada máquina tiene
    #un peso distinto. El resultado de esta suma es el valor de nuestra función objetivo para
    #la secuencia dada

    suma_waiting_machines=[]
    funcion_objetivo=0
    sumapormaquina=0
    for i in range(0,len(waiting_machines)):
        for j in range(0,len(waiting_machines[i])):
            sumapormaquina=sumapormaquina+waiting_machines[i][j]
        suma_waiting_machines.append(sumapormaquina)
        funcion_objetivo=funcion_objetivo+suma_waiting_machines[i]*pesos_maquinas[i]
        sumapormaquina=0
    return funcion_objetivo


# Función para crear una secuencia aleatoria Jobshop.(También se podía haber hecho con
random_solution()) :

def secuencia_aleatoria_jobshop(instance):
    lista=[]
    for i in range(0,instance.jobs):
        for j in range(0,instance.machines):
            lista.append(i)
    random.shuffle(lista)
    return lista

#Función que calcula el fitness de una población:

def calcula_fitness_poblacion(instance,poblacion):
    fitness_poblacion=[]
    for i in range(0,len(poblacion)):
        fitness_poblacion.append(FUNCION_OBJETIVO(instance,poblacion[i]))
    return fitness_poblacion

# Función que crea la población inicial del algoritmo, para su funcionamiento,
# necesita la función anterior --> secuencia_aleatoria_jobshop() :

def creacion_poblacion_inicial(instance,tamaño_poblacion):
    poblacion=[]
    for i in range(0,tamaño_poblacion):
        poblacion.append(secuencia_aleatoria_jobshop(instance))
    return poblacion

#Funcion que calcula el cruce de dos secuencias mediante método POX:

def cruce_POX(instance,parent1,parent2):
    #Seleccion de subconjuntos jobset1 y jobset2 creado de forma random
    offspring1=[]
    offspring2=[]
    jobset1=[]
    jobset2=[]
    jobset1 = random.sample(range(instance.jobs),random.randint(1,instance.jobs-1))
    [jobset2.append(x) for x in range(0,instance.jobs) if x not in jobset1]

    #Creamos los descendientes : offspring1 y offspring2
```

```
    for i in range(0,len(parent1)):
        if parent1[i] in jobset2:
            offspring2.append(parent1[i])
        if parent2[i] in jobset2:
            offspring1.append(parent2[i])
    for i in range(0,len(parent1)):
        if parent1[i] in jobset1:
            offspring1.insert(i,parent1[i])
        if parent2[i] in jobset1:
            offspring2.insert(i,parent2[i])
    return offspring1,offspring2

#Función que intercambia dos posiciones de una secuencia:

def intercambio(instance,secuencia):
    s=random.sample(range(0,len(secuencia)),2)
    secuencia_nueva=copy.deepcopy(secuencia)
    aux=secuencia_nueva[s[0]]
    secuencia_nueva[s[0]]=secuencia_nueva[s[1]]
    secuencia_nueva[s[1]]=aux
    return secuencia_nueva

#Función que lleva a cabo insertion dada dos posiciones:

def insertion(secuencia,posicion_1,posicion_2):
    aux=secuencia[posicion_1]
    secuencia.pop(posicion_1)
    secuencia.insert(posicion_2,aux)
    return secuencia

#Función que compara dos secuencias y devuelve la que tiene mejor fitness:

def comparacion_secuencias(instance,mejorsecuencia,secuencia2):
      mejorvalor=FUNCION_OBJETIVO(instance,mejorsecuencia)
      valor_sec2=FUNCION_OBJETIVO(instance,secuencia2)
      if(valor_sec2<mejorvalor):
          mejorvalor=valor_sec2
          mejorsecuencia=copy.deepcopy(secuencia2)
      return mejorvalor,mejorsecuencia

#función que devuelve la mejor secuencia de una población:

def mejorsec_poblacion(instance,poblacion):
    #mejorsecuencia=[]
    mejorsecuencia=copy.deepcopy(poblacion[0])
    for i in range(1,len(poblacion)):
        mejorvalor,mejorsecuencia=comparacion_secuencias(instance,mejorsecuencia,poblacion[i])
    return mejorsecuencia,mejorvalor

#-------------------------------------------------------------------------------
''' 3) Este apartado equivale al "main", ejecución del algoritmo Genético'''

### ---------------------- MAIN - EJECUCIÓN DE ALGORITMO  ------------------- ###

for i in range(2):

    archivo='test_jobshop_ft'+ str(i+1)+'pru.txt'
    lista_exp=[]
```

```
    lista_exp_2=[]

    for exp in range(0,10):

        #Inicio de tiempo de ejecución:
        start=time.time()

        #Carga de la instancia Jobshop:
        instancia = JobShop(archivo)

        #Definición del tamaño de la población (population size):
        tamaño_poblacion=10

        #Creación de una población inicial con secuencias random:
        poblacion=creacion_poblacion_inicial(instancia,tamaño_poblacion)

        #obtención de la secuencia con mejor fitness de la población:
        bestseq_pob,fitness_best=mejorsec_poblacion(instancia,poblacion)

        end=time.time()
        tiempo_ejecucion=end-start
        while(tiempo_ejecucion<15):
            parent1=poblacion[random.randint(0,len(poblacion)-1)]
            parent2=copy.deepcopy(bestseq_pob)
            child_1,child_2=cruce_POX(instancia,parent1,parent2)
            child_1_mut=intercambio(instancia,child_1)
            child_2_mut=intercambio(instancia,child_2)

mejorvalor_child_mut,mejor_child_mut=comparacion_secuencias(instancia,child_1_mut,child_2_mut)
            fitness_child=FUNCION_OBJETIVO(instancia,mejor_child_mut)
            for i in range(0,len(poblacion)):
                if fitness_child<(FUNCION_OBJETIVO(instancia,poblacion[i])):
                    poblacion.pop(i)
                    poblacion.insert(i,mejor_child_mut)
            if fitness_child<fitness_best:
                fitness_best=fitness_child
                bestseq_pob=copy.deepcopy(mejor_child_mut)
            end=time.time()
            tiempo_ejecucion=end-start

        lista_exp.append(fitness_best)
        lista_exp_2.append(instancia.Cmax(bestseq_pob))

    with open('resultados_AG_FO_t15_p10.txt','a') as archivo_resultados:
        archivo_resultados.write(archivo + ' ' + 'F.O' + str(lista_exp) + ' ' + '\n')
        archivo_resultados.write(archivo + ' ' + 'MAKESPAN' + str(lista_exp_2) + ' ' + '\n')
        archivo_resultados.write('\n')


for i in range(30):

    archivo='test_jobshop_LA'+ str(i+1)+'.txt'
    lista_exp=[]
    lista_exp_2=[]

    for exp in range(0,10):
```

```
        #Inicio de tiempo de ejecución:
        start=time.time()

        #Carga de la instancia Jobshop:
        instancia = JobShop(archivo)

        #Definición del tamaño de la población (population size):
        tamaño_poblacion=10

        #Número de iteraciones del algoritmo (Condición de parada):
        #Num_iter=3000

        #Creación de una población inicial con secuencias random:
        poblacion=creacion_poblacion_inicial(instancia,tamaño_poblacion)

        #obtención de la secuencia con mejor fitness de la población:
        bestseq_pob,fitness_best=mejorsec_poblacion(instancia,poblacion)

        end=time.time()
        tiempo_ejecucion=end-start
        while(tiempo_ejecucion<15):
            parent1=poblacion[random.randint(0,len(poblacion)-1)]
            parent2=copy.deepcopy(bestseq_pob)
            child_1,child_2=cruce_POX(instancia,parent1,parent2)
            child_1_mut=intercambio(instancia,child_1)
            child_2_mut=intercambio(instancia,child_2)

mejorvalor_child_mut,mejor_child_mut=comparacion_secuencias(instancia,child_1_mut,child_2_mut)
            fitness_child=FUNCION_OBJETIVO(instancia,mejor_child_mut)
            for i in range(0,len(poblacion)):
                if fitness_child<(FUNCION_OBJETIVO(instancia,poblacion[i])):
                    poblacion.pop(i)
                    poblacion.insert(i,mejor_child_mut)
            if fitness_child<fitness_best:
                fitness_best=fitness_child
                bestseq_pob=copy.deepcopy(mejor_child_mut)

            end=time.time()
            tiempo_ejecucion=end-start

        lista_exp.append(fitness_best)
        lista_exp_2.append(instancia.Cmax(bestseq_pob))

    with open('resultados_AG_FO_t15_p10.txt','a') as archivo_resultados:
        archivo_resultados.write(archivo + ' ' + 'F.O' + str(lista_exp) + ' ' + '\n')
        archivo_resultados.write(archivo + ' ' + 'MAKESPAN' + str(lista_exp_2) + ' ' + '\n')
        archivo_resultados.write('\n')
```