Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

UNIVERSIDAD DE VALLADOLID
ESCUELA DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería de Organización Industrial

Técnicas de optimización para la resolución del problema del Job Shop

Autor:

Sanz Torrero, Elena

Tutor:

Pérez Rodríguez, María Teresa
Departamento Matemática Aplicada

Valladolid, julio de 2016.

Agradecimientos

A mi tutora, María Teresa Pérez, por su ayuda y tiempo dedicado, sin lo cual este trabajo no hubiera sido posible.

A mi familia y amigos que me han acompañado en esta etapa y en especial a Ismael por su apoyo incondicional.

Por último, a mi abuela Juliana por su amor y generosidad.

Resumen:

En el presente trabajo se llevará a cabo el estudio del problema *del job shop*, catalogado dentro de la rama de optimización combinatoria y más concretamente de programación de tareas. Para su estudio y resolución se han empleado técnicas metaheurísticas, en concreto tres: Algoritmos Genéticos, Algoritmo de Colonia Artificial de Abejas y Búsqueda Tabú. Para cada uno de estos métodos se incluye una explicación del mismo, su estructura paso a paso aplicada al *job shop* y la resolución de un ejemplo de un problema concreto para visualizar mejor el funcionamiento del algoritmo. Adicionalmente, para la Búsqueda Tabú se ha llevado a cabo su implementación en el programa Matlab.

Palabras clave: "Problema del job shop", "Algoritmos genéticos", "Algoritmo Colonia de Abejas", "Búsqueda Tabú", "*Métodos heurísticos*"

Abstract:

The following Project studies the Job Shop Problem, included in the combinatorial optimization area, more concretely in task scheduling. For the study and resolution of this problem, several metaheuristic techniques have been used, focusing on three of them: Genetic Algorithms, Tabu Search and Artificial Bee Colony Algorithm. For each one of these methods it has been included a theoretical explanation, its own structure applied to the Job Shop Scheduling and the resolution of a specific example of the problem in order to prove the performance of the algorithm. Furthermore (In addition) the Tabu Search has been implemented in the Matlab Program.

Key words: "Job Shop Scheduling Problem", "Genetic Algorithms", "Tabu Search", "Simulated-Artificial Bee Colony Algorithm", "Heuristic techniques"

Índice general

Introducción

En el presente trabajo se estudiará el problema del *job shop*, también conocido por su nombre en inglés *Job Shop Scheduling Problem* (JSSP), planteando la aplicación de diferentes métodos de optimización para su resolución, especialmente métodos heurísticos.

El documento se estructura en cuatro capítulos diferenciados, excluyendo el presente apartado de introducción, y la parte de conclusiones y anexos.

El primer capítulo está dedicado al problema de estudio, el *job shop*, en él se dará una definición detallada y se verán las distintas formas de representarlo, además de hablar sobre su historia, dificultad, aplicaciones y métodos utilizados en su resolución, entre otros.

Los tres capítulos siguientes están centrados cada uno en un método de resolución diferente y su aplicación al problema. La estructura de los tres capítulos es muy similar, introduciendo primero el método para después detallar su estructura y explicar paso a paso su funcionamiento. Además, al final del capítulo se ilustra lo comentado anteriormente a través de la resolución de un ejemplo de un problema concreto del *job shop*, mediante la técnica en cuestión.

Los tres métodos desarrollados son algoritmos genéticos, algoritmo de colonia artificial de abejas y búsqueda tabú, correspondientes a los capítulos dos, tres y cuatro respectivamente.

Adicionalmente, el capítulo que aborda la búsqueda tabú incluye un apartado dedicado a la exposición de los resultados computacionales obtenidos mediante dicho algoritmo programado en Matlab. El código del programa mencionado se puede encontrar en los anexos del presente trabajo, en los cuales se detallará su funcionamiento.

Finalmente, el último capítulo recoge las conclusiones del trabajo, comentando algunas posibles líneas futuras por las que continuar la investigación.

Motivación y objetivos

Los problemas de programación de tareas aparecen con frecuencia en la vida real y se basan en la organización de una serie de trabajos que comparten un conjunto limitado de recursos cumpliendo diversas restricciones. El problema consiste en la asignación de dichos recursos a las tareas con el objetivo de optimizar un determinado criterio, normalmente relacionado con el tiempo o coste del proceso.

Se pueden encontrar aplicaciones de este tipo de problemas en áreas muy diferentes, como la producción, la logística u aplicaciones computacionales entre otros. Su uso más frecuente en la industria.

Estos problemas son considerados como muy complejos, por lo que su resolución mediante métodos exactos resulta muy costosa. Para ello se utilizan técnicas heurísticas que simplifican la búsqueda sin necesidad de analizar una a una todas las posibles soluciones.

En el presente trabajo, como ya se ha comentado, dentro de la categoría de programación de tareas se estudiará el problema del *job shop*, considerado como uno de los de mayor complejidad, por su elevado número de posibles soluciones, el cual crece exponencialmente en caso de aumentar las dimensiones del problema.

Es por ello, por lo que se hace muy necesario emplear técnicas de optimización, en concreto en este trabajo se plantean tres métodos heurísticos: algoritmos genéticos, algoritmo de colonia artificial de abejas y búsqueda tabú.

El objetivo del proyecto será presentar un amplio estudio sobre el problema y desarrollar las técnicas citadas anteriormente primero de forma general y luego su aplicación concreta al problema del *job shop*, detallando paso a paso su funcionamiento.

Capítulo 1

El Problema del Job Shop.

1.1. Introducción.

El problema del *job shop*, también conocido por JSSP de sus siglas en inglés *Job Shop Scheduling Problem*, es un problema de optimización combinatoria y más concretamente de programación de tareas, además dentro de esta categoría, el *job shop* es uno de los que más interés despierta con un gran número de estudios que lo abordan.

El problema del *job shop* clásico se describe como sigue: se tiene un conjunto de trabajos que necesitan ser procesados, cada uno a su vez formado por un conjunto de operaciones. Para ello, se dispone de un conjunto de máquinas que realizan diversas funciones. Las operaciones de los trabajos van pasando sucesivamente por el conjunto de máquinas, de forma que cada una de ellas tiene asignada una máquina en la que ser ejecutada, además de un tiempo de proceso. Las operaciones que forman un trabajo, el orden que ocupan en la secuencia del trabajo, la máquina asignada y el tiempo que tarda en ejecutarse la operación vienen dados por el problema y no varían.

El dato que sí cambia y es objeto de estudio del problema, es el orden en el que las distintas operaciones pasan por el conjunto de máquinas, o lo que es lo mismo, la secuencia de procesamiento de las operaciones en las máquinas. Llamamos *programa* a la asignación fija a cada operación de un intervalo de tiempo para ser procesada. Por tanto, el programa será una solución factible del problema. El objetivo es encontrar el programa (solución) que optimice un determinado criterio.

Los objetivos pueden ser muy diversos, el más común en este tipo de problemas es el de minimizar el *makespan*, siendo el *makespan* el tiempo necesario para completar todas las operaciones. Aunque también podemos encontrar otra serie de objetivos como por ejemplo, dado un coste asociado a

cada operación, minimizar el coste total de proceso, o dada una fecha máxima de entrega de cada trabajo, minimizar el retraso máximo. Esto se verá más detallado en el apartado 1.6 de éste capítulo en el que se habla de las funciones objetivo.

El problema del *job shop* tiene muchas aplicaciones en la industria, además de ser probablemente el área en el que más se aplica, en éste ámbito menos general se considera al *job shop* un sistema de producción o proceso productivo. Relacionado con este campo, se encuentra el problema del *flow shop*, caso particular del *job shop*, pero con la diferencia de que en el *flow shop*, todos los trabajos tienen la misma secuencia de máquinas, es decir, el orden en el que van pasando por las distintas máquinas es el mismo para todos los trabajos, habiendo que decidir únicamente el orden de los trabajos en el que entran al proceso.

A continuación se explicarán los aspectos más importantes del job shop como su dificultad computacional, su origen o sus aplicaciones entre otros, además de dar una definición más completa del problema.

1.2. Dificultad.

La teoría de la complejidad computacional realiza una distinción entre varias clases de problemas de optimización combinatoria [8]:

Problemas tipo P (Polinómicos): existe al menos un algoritmo polinómico para resolver el problema, que proporciona soluciones en un tiempo computacional razonable. Se entiende por algoritmo polinómico o de tiempo polinomial aquel con función de complejidad temporal en O(p(n)) para alguna función polinómica *p*, donde *n* denota el tamaño de la entrada [72].

Problemas tipo NP (Polinómicos no determinísticos): aún no se ha encontrado un algoritmo eficiente que los resuelva aunque puede aplicarse un algoritmo polinómico para comprobar si una posible solución es válida o no. Esta característica permite métodos de resolución donde se van aceptando o desestimando soluciones hipotéticas.

Problemas NP-hard (o NP-duros): son problemas de extrema complejidad, son equivalentes entre sí. Se cree que no existen algoritmos polinómicos para resolverlos, aunque esto no se ha demostrado.

En la siguiente imagen se muestra un diagrama con los tipos de problemas. Se puede decir que los problemas de NP-completo son los problemas más difíciles de NP y se supone que NP≠P, aunque no se ha demostrado.

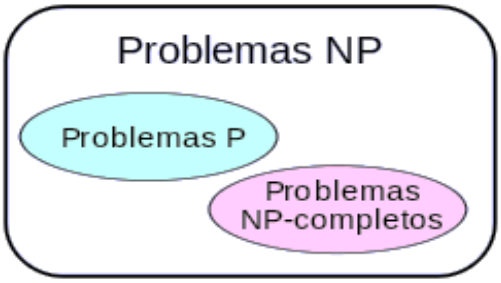


Diagrama de los tipos de problemas en función de su complejidad.

La dimensionalidad del problema del *job shop* viene dada por *n x m*, siendo *n* el número de trabajos y m el número de máquinas, de tal forma que se tienen al menos $(n!)^m$ posibles soluciones. Cuanto mayor sean tanto *n* como *m*, el número de soluciones crece exponencialmente. Esta característica es lo que hace que el problema sea considerado como NP-hard [52], M. R. Garey y D. S. Jonhson [42], demostraron que el JSSP es un problema NP-hard, y de esa clase, es uno de los menos tratables.

Para hacernos una idea de la dimensión del problema, pongamos un ejemplo en el que tenemos 24 operaciones y una máquina, las posibles soluciones serían (24!) = 6.20 x 10^{23}. Suponemos que la edad del Universo es aproximadamente 14 mil millones de años = 4.41 x 10^{17} segundos = 4.41x10^{23} μsegundos. Si tuviéramos un ordenador capaz de analizar una solución cada μsegundo, aun considerando toda la edad del Universo apenas se podría dar solución al problema por enumeración (analizar una a una todas las soluciones posibles) [22].

Por lo que para un problema que a simple vista no parecía excesivamente largo o complejo (24 operaciones y una máquina), analizar todas las soluciones se convierte en algo totalmente inabarcable.

1.3. Historia [28, 17].

La historia del problema del Job Shop comienza hace más de 60 años y no está muy claro a quién atribuir el primer planteamiento del problema.

Una de las primeras publicaciones relacionadas data de 1954, en la que S. M. Johnson [43] formula el algoritmo del *flow shop*, que poco después, en 1956 J. R. Jackson [41] generalizaría al algoritmo del *job shop*.

Asimismo en 1955, S. B. Akers y J. Friedman [3] presentan un enfoque geométrico para resolver un problema *job shop* con dos trabajos. Más tarde, aparecerán diversas publicaciones basadas en dicho enfoque, entre las que se encuentran los trabajos de P. Brucker (1988) [19], Y. N. Sotskov (1985) [64] o W. Szwarc (1960) [66].

Sin embargo fue gracias a la introducción por parte de H. Fisher y G. L. Thompson (1963) [29] de dos problemas de referencia: uno con 10 trabajos y 10 máquinas y otro con 20 trabajos y 5 máquinas, cuando el problema del *job shop* crece en popularidad, ya que estos problemas suponen un reto para los investigadores. Los primeros intentos de dar solución a dichos problemas vienen de G. H. Brooks y C. R. White (1965) [18] y H. H. Greenberg (1968) [37], cuyos métodos se basan en programación entera. Más tarde otros investigadores como E. Balas (1969) [12], J. M. Charlton y C. C. Death (1970) [23], M. Florian et al. (1971) [31], Ashour y Hiremath (1977) [11] y Fisher (1973) [30] probaron suerte con Multiplicadores Lagrangianos.

Mientras que para el problema 20 trabajos-5 máquinas se tardó diez años en dar con la solución óptima, para el problema 10 trabajos-10 máquinas fueron necesarios veinticinco años. En 1984, Lageweg encontró la solución óptima, pero sin demostrar que lo era. Esta demostración llegó en 1989 de la mano de J. Carlier y E. Pinson [21]. Otros autores que consiguieron resolver el problema 10x10 junto con otros pequeños problemas fueron D. Applegate y W. Cook (1991) [7], P. Brucker et al. (1994) [20], D. P. Williamson et al. (1997) [73] mediante el desarrollo de métodos exactos. Aunque, hasta el momento los métodos exactos no pueden resolver problemas de 15x15, siendo necesarios métodos heurísticos.

En el mismo año de la publicación de Fisher y Thompson (1963), se edita el libro “Industrial Scheduling” de J. F. Muth y G. L. Thompson [49] que se considera la base de la mayoría de las investigaciones posteriores.

En 1964, B. Roy y B. Sussman [62] propusieron por primera vez la representación del problema mediante un grafo disyuntivo y, en 1969, E. Balas [12] fue el primero en aplicar un procedimiento enumerativo basado en dicho grafo.

Como se ha comentado anteriormente, la resolución mediante métodos exactos no da resultados para problemas de gran dimensión, por lo que las investigaciones toman una nueva dirección hacia los métodos heurísticos o de aproximación. Son muchas las publicaciones que usan estas técnicas.

Una de las primeras técnicas, dentro de los métodos de aproximación, fue la heurística de cuello de botella, plasmada en los estudios de J. Adams et al. (1988) [1], E. Balas et al. (1995) [13] o S. Dauzere-Peres y J. B. Lasserre (1993) [25].

Asimismo, los algoritmos de búsqueda local se volvieron muy populares; estos algoritmos se basan en una estructura de entornos, en los que las nuevas soluciones se obtienen a partir de las existentes. Podemos ver los resultados de su aplicación en las publicaciones de C. A. Glass et al. (1992) [32], E. J. Anderson et al. (1995) [6] y R. J. P. Vassens et al. (1995) [69].

Paralelamente, se comenzaron a aplicar técnicas más potentes, como el recocido simulado [70] (o *simulated annealing* en inglés), la búsqueda tabú de la mano de M. Dell`Amico et al. (1993) [27] o la excelente implementación de la misma por parte de Nowicki y Smutnicki (1993) [50] y Balas y Vazacopulos (1995) [14].

Asimismo, T. Yamada et al. (1992) [75], D. C. Mattfeld (1995) [48] entre otros probaron suerte con los algoritmos genéticos.

Tanto las técnicas de búsqueda local, como los algoritmos genéticos son bastante robustos y no requieren demasiado trabajo de implementación. Esto los hace buenos métodos para resolver el problema del *job shop*.

El método meteheurístico GRASP se aplicó por primera vez al problema a principios de los años 2000, como ejemplos tenemos las publicaciones de Binato et al. (2002) [16], Aiex et al. (2003) [2], o Ravetti et al. (2007) [60].

En el apartado de este trabajo de métodos de resolución, se hablará con más detalle de las técnicas comentadas anteriormente.

1.4. Aplicaciones

En general, todos los problemas de programación de tareas (scheduling), entre los cuales se encuentra el JSSP, se basan en la asignación en el tiempo de recursos a una serie de tareas (satisfaciendo algunas restricciones). Normalmente estos recursos son limitados y por ello el objetivo es optimizar dicha asignación.

Los recursos, las tareas e incluso los objetivos pueden tomar muchas formas [36]:

Los recursos pueden ser máquinas, trabajadores, herramientas, medios informáticos, puertas de embarque en un aeropuerto, etc., en general cualquier elemento disponible para resolver una necesidad.

Como ejemplos de tareas, tenemos las fases de producción de cualquier bien material, las diversas etapas de un proyecto, los despegues y aterrizajes de un avión, etc. Las tareas normalmente tienen asignada una restricción de precedencia entre ellas y también pueden tener una prioridad, una fecha de inicio o una fecha límite para ser ejecutadas.

Respecto a la optimización del proceso, los objetivos buscados varían en función de las necesidades. Uno de los objetivos más comunes es el de minimizar el *makespan* o tiempo total en el que se completan todas las operaciones, pero también podemos encontrar otros como minimizar el coste de producción, el retraso de las tareas, etc.

A continuación se citan algunos ejemplos de aplicaciones sobre programación de tareas en la vida real que recoge M. L. Pinedo en su obra [59]:

Industria

Las aplicaciones más directas y frecuentes las encontramos en el sector de la industria. En la fabricación de cualquier bien material intervienen varias operaciones para transformarlo desde su forma original hasta el producto terminado. Lo que se busca es que esto se haga en el mínimo tiempo posible, que los productos estén terminados antes de su fecha de entrega o que se aprovechen al máximo todos los recursos. Seguidamente detallamos algunos ejemplos:

Fabricación de semiconductores

El proceso de producción de semiconductores normalmente lo forman cuatro fases: fabricación de la oblea, sondeo de la oblea, ensamblaje y prueba final. Las obleas están compuestas por diversas capas y su fabricación consta a su vez de varias operaciones (limpieza, oxidación, deposición y metalización, litografía…) que hay que repetir para cada una de las capas. Esto supone un alto nivel de recirculación en el proceso. Si además añadimos que el número de pedidos de semiconductores suele ser en torno a la centena, y cada uno tiene su propia fecha de comienzo y de entrega, al final tenemos un proceso muy complejo con un gran número de operaciones a planificar.

El objetivo buscado suele ser cumplir los plazos de entrega, maximizando el rendimiento y aprovechando el uso de los equipos, especialmente en las zonas de cuello de botella.

Línea de montaje de automóviles

Una cadena de montaje de automóviles habitualmente fabrica modelos muy diversos de vehículos: más grandes, más pequeños, con dos o cuatro puertas, de colores muy variados y con diferentes prestaciones. Por tanto no todos los coches siguen las mismas operaciones, lo cual complica el proceso. Donde más se evidencia este hecho es en la fase de pintura, dado que si se quiere cambiar de color es necesario un tiempo de preparación de la máquina.

Una de las cosas que se pretende con una buena planificación de tareas es evitar los llamados cuellos de botella y mantener un equilibrio de trabajo entre todas las máquinas.

Logística

Dentro del campo de la logística encontramos multitud de aplicaciones, aunque aquí solamente se citan dos casos concretos sobre aplicación en medios de transporte. Tendríamos más ejemplos en otras áreas como son las

relacionadas con planificación de horarios, programación de torneos deportivos, etc.

Como se puede observar, este campo está íntimamente relacionado con el sector servicios. Éste se caracteriza por un trato más directo con el cliente, y la variación de los recursos (en contraposición a una fábrica por ejemplo, en la que el número y tipo de máquinas son fijos).

Asignación de puertas de embarque en un aeropuerto

En este caso consideramos como *recursos* las puertas de embarque, y las *tareas* serían el uso que hacen de ellas los aviones, de esta forma, una tarea comienza cuando aterriza un avión y finaliza cuando despega.

Existen numerosos factores a tener en cuenta: por un lado, las puertas de embarque no están preparadas para todos los tipos de aviones y por otro, los aviones en teoría tienen un calendario fijo de llegada y salida, pero en la práctica, en dicho calendario influyen muchos factores aleatorios como el clima, la disponibilidad del aeropuerto destino (por lo que el avión no podrá despegar hasta que se compruebe que éste estará libre a su llegada), etc.

La planificación de ese proceso busca asignar físicamente cada avión a una puerta de embarque de manera que se minimicen los retrasos y se aprovechen adecuadamente todos los recursos del aeropuerto, tanto materiales como humanos.

Planificación de horarios ferroviarios

Este caso de estudio del tipo Job Shop lo podemos encontrar en el trabajo de Ingolotti Hetter [40]. El problema trata de asignar un horario factible a cada tren teniendo en cuenta su ruta y una serie de restricciones (número disponible de vías, frecuencia de salida entre trenes, máximo retraso permitido, cierre de las instalaciones para mantenimiento…).

Análogamente, los recursos son las vías de la estación y las tareas son el paso de un tren por la estación, siendo un trabajo el recorrido completo del tren.

Éste es un problema especialmente complejo ya que intervienen un elevado número de factores que es necesario considerar. El objetivo en este caso es encontrar un horario factible para cada ruta de forma que se minimicen los retrasos y se aprovechen adecuadamente los recursos.

Se ha detallado un caso concreto, pero en relación a la planificación de horarios existen un sinfín de ejemplos como: asignación de turnos de trabajadores (médicos y enfermeros en un hospital, operarios en una cadena de montaje…), calendario de vuelos de un aeropuerto, horarios y salas en unos grandes cines, etc.

Otras aplicaciones

Además de los ejemplos ya comentados, podemos encontrar técnicas de programación de tareas (aunque en menor medida) en otros ámbitos como por ejemplo:

Planificación de tareas en la CPU:

Un sistema operativo como la CPU ejecuta las tareas de los diferentes programas y aplicaciones del ordenador. La meta perseguida es conseguir que las tareas se ejecuten en el menor tiempo posible, ya que eso incrementa las prestaciones, para ello es indispensable una buena elección de la planificación.

1.5. Formulación y representación.

1.5.1. Formulación matemática.

Definición: sean dados un conjunto J de trabajos y un conjunto M de máquinas, cada trabajo J_j consta de un conjunto O_j de operaciones que deben ser ejecutadas de forma secuencial en las distintas máquinas. Por tanto, para cada operación O_{ij} tenemos asociado un trabajo J_j al que pertenece y una máquina M_h en la que debe ser procesada consumiendo un tiempo p_{ij} (ininterrumpido y positivo). Además, el conjunto O que contiene todas las operaciones del problema, se descompone en cadenas (una para cada trabajo) que marcan una relación de precedencia binaria entre operaciones. El objetivo es encontrar una asignación de tiempos de inicio para cada operación, de tal forma que se minimice el *makespan,* definido como el tiempo requerido para acabar todos los trabajos.

Notación:

J_j: trabajo, tal que j = 1… n.

M_h: máquina, tal que h = 1 … m.

O_{ij}: operación i del trabajo j, tal que i = 1 … k_j, siendo k_j el número de operaciones del trabajo j.

h_{ij}: máquina de ejecución de la operación O_{ij}.

p_{ij}: duración de la operación O_{ij}.

b_{ij}: fecha de comienzo de la operación O_{ij}.

c_{ij}: fecha de finalización de la operación O_{ij}, tal que $c_{ij} = b_{ij} + p_{ij}$.

c_{max}: *makespan*, tal que $c_{max} = \max\{c_j\}$.

D_j: fecha de entrega comprometida del pedido *j* (due date).

r_j: fecha más temprana desde la que está disponible el trabajo *j*.

El problema está sujeto a las siguientes restricciones [53]:

-Una operación no puede pasar dos veces por la misma máquina.

-No hay relaciones de precedencia entre operaciones de distintos trabajos.

-Las operaciones no se pueden interrumpir.

-Ninguna máquina está disponible antes de t=0.

-Cada máquina sólo puede ejecutar una tarea a la vez.

-Existe una relación de precedencia fija entre las operaciones de un mismo trabajo.

1.5.2. Representación mediante grafo.

Llamamos **grafo** al par formado por: G = (V, E), siendo:

V: conjunto finito de vértices o nodos.

E: conjunto de arcos o aristas que conectan los nodos entre sí, de tal forma que E ⊂ V x V.

El problema del JSSP puede ser representado mediante un grafo disyuntivo formado por tres conjuntos:

V: representaremos las **operaciones** mediante el conjunto de vértices o nodos, este conjunto incluirá además dos operaciones ficticias que por lo tanto no están asociadas a ninguna máquina ni trabajo y su tiempo de duración es nulo. Estas dos operaciones representan el inicio y final de la producción, es lo que se llama *nodo fuente* y *nodo sumidero* respectivamente.

Las relaciones de precedencia entre operaciones las designamos mediante los arcos del grafo, aunque hay que tener en cuenta que tenemos dos tipos distintos de precedencias:

A: conjunto de arcos dirigidos que refleja las **precedencias tecnológicas** entre las operaciones pertenecientes a un mismo trabajo. Asimismo, conecta el nodo fuente y el nodo sumidero a las primeras y últimas operaciones de cada trabajo respectivamente.

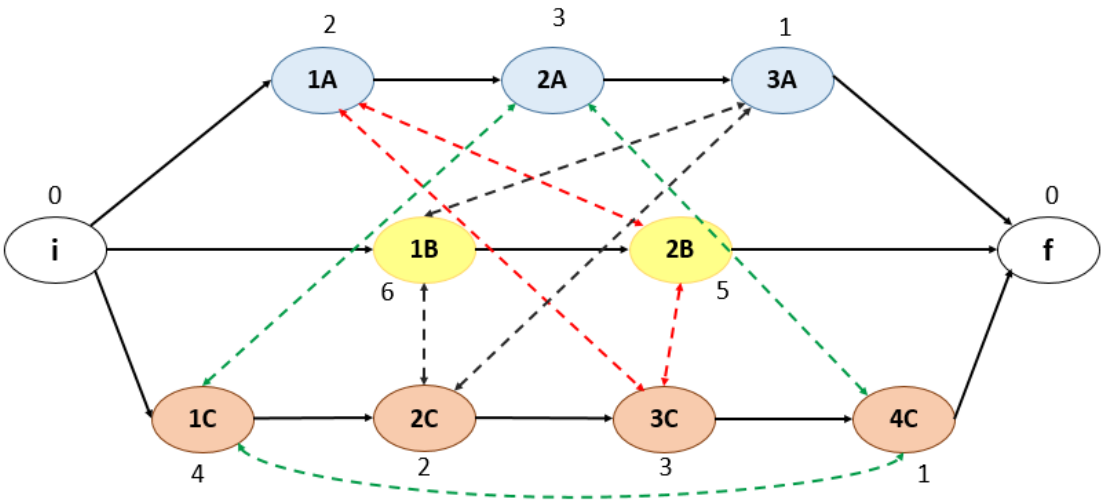
S: conjunto de arcos no dirigidos que une todas las operaciones que tienen lugar en la misma máquina.

El resultado es un grafo disyuntivo: G = (V,A U S).

Para ilustrar esto, veamos un ejemplo:

TRABAJO	OPERACIÓN	TIEMPO EJECUCIÓN	MÁQUINA
A	1A	2	M1
	2A	3	M2
	3A	1	M3
B	1B	6	M3
	2B	5	M1
C	1C	4	M2
	2C	2	M3
	3C	3	M1
	4C	1	M2

La representación de este problema como un grafo disyuntivo sería:



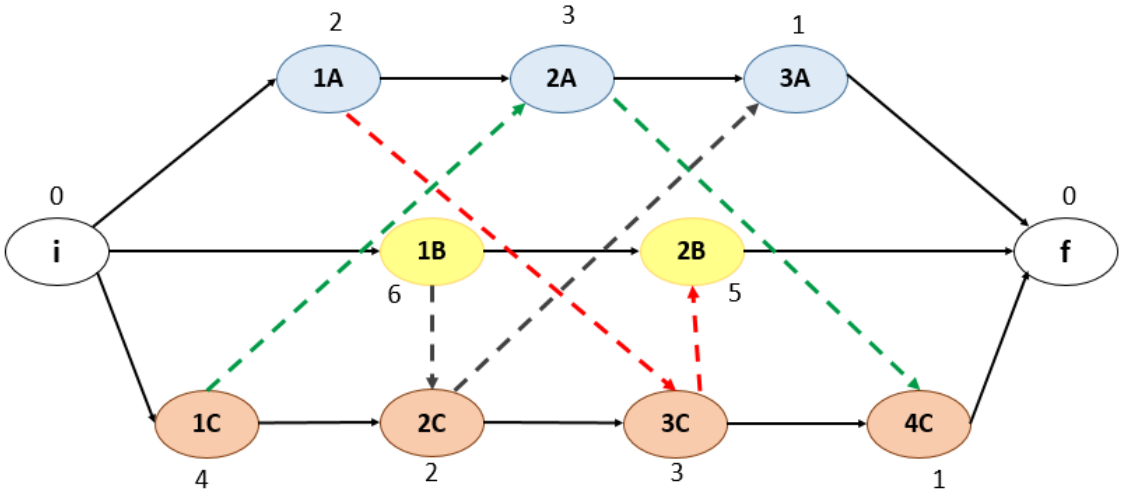
Como se ve en la figura y acorde al enunciado, las operaciones 1A, 2A y 3A pertenecen al trabajo A, y deben ser ejecutadas en ese mismo orden (precedencia tecnológica), asimismo las operaciones 1A, 2B y 3C pertenecientes cada una a un trabajo diferente, se ejecutan todas en la máquina 1, y están unidas mediante arcos no dirigidos ya que el orden en que las operaciones pasan por las máquinas aún no se ha determinado. Y de la misma forma para los demás trabajos y máquinas.

En la imagen, los arcos que indican la precedencia entre operaciones de un mismo trabajo (de color negro) son dirigidos, mientras que los arcos de las precedencias entre máquinas (en líneas discontinuas y de colores: rojo para la

máquina 1, verde para la máquina 2 y azul para la máquina 3) son no dirigidos o disyuntivos.

Como se puede observar, los tiempos de ejecución de las operaciones, aparecen en su nodo correspondiente. Siendo dichas duraciones nulas en los nodos inicial y final. Existe también otra forma de representarlo en el grafo, en la que los tiempos de ejecución se indican en las aristas salientes de cada nodo. Así, por ejemplo para la actividad 1A, cuya duración es 2, cada una de sus aristas salientes tendrían valor 2.

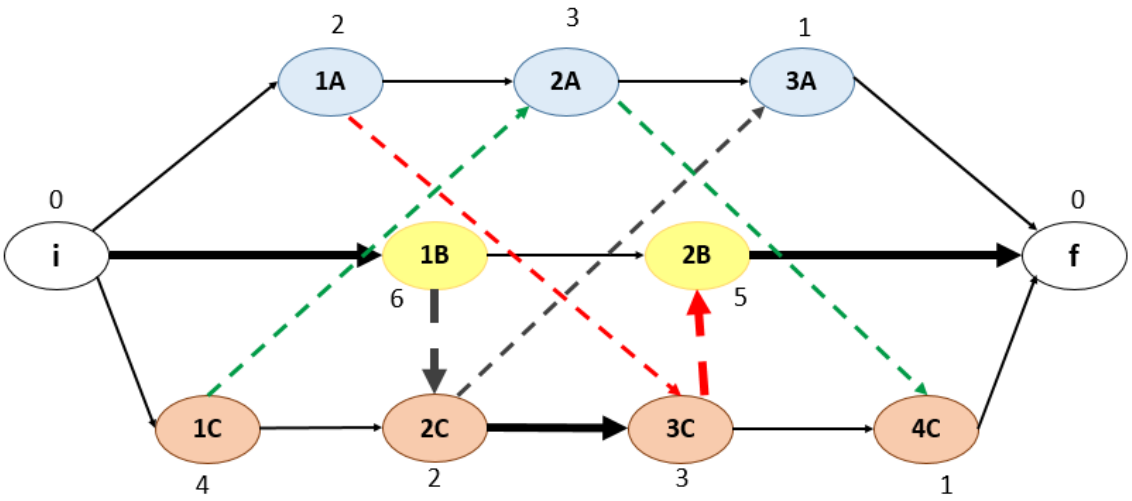
Llamamos *programa* a la asignación a cada operación de un intervalo de tiempo para ser efectuada, o lo que es lo mismo, dar un orden de procesamiento de las operaciones en la máquina. En la representación mediante grafo, esto se consigue dotando de dirección a los arcos disyuntivos. Para que una solución sea factible, la selección de los arcos disyuntivos debe ser *completa* (todos los arcos deben tener una dirección determinada) y *consistente* (esto se consigue si el grafo resultante es acíclico) [75].



En la imagen vemos un ejemplo de programa para el problema planteado. El programa es el siguiente: para la máquina 1: 1A->3C->2B, para la máquina 2: 1C->2A->4C, y para la máquina 3: 1B->2C->3A. Todos los arcos son ahora dirigidos.

El objetivo buscado es que dicha solución o programa tenga el mínimo *makespan*, que como ya se ha comentado previamente, es el tiempo en el que se ejecutan todos los trabajos. En relación a los grafos, el *makespan* coincide con la longitud del camino crítico, es decir, el camino más largo que va desde el nodo inicial hasta el nodo final [15].

A continuación se muestra el camino crítico para el programa visto, marcados sus arcos en líneas gruesas. El camino crítico, en este caso lo forman las operaciones: 1B, 2C, 3C, 2B.



1.5.3. Representación mediante diagrama de Gantt.

En apartados anteriores hemos visto cómo representar el problema del *job shop* y sus programas correspondientes, pero quizá la representación más intuitiva que existe es la representación mediante el diagrama de Gantt.

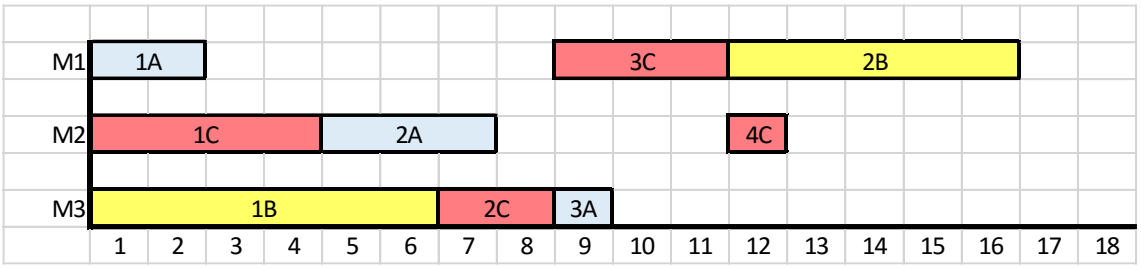
Entendemos que el diagrama de Gantt es una herramienta en forma de gráfico de barras, usada para planificar y programar tareas a lo largo de un determinado periodo de tiempo. Gracias a una fácil visualización de las operaciones a realizar, permite el control del proceso. Reproduce gráficamente las tareas, su duración y secuencia, además de la fecha de finalización de cada una de ellas [51].

Las operaciones que hay que programar tienen asignada una máquina concreta en la que ser procesadas. Por tanto, nuestro diagrama de Gantt tendrá en el eje de ordenadas cada una de las máquinas que se utilizan, mientras que el tiempo estará indicado en el eje de abscisas.

A continuación se expone un ejemplo (el mismo visto anteriormente) de un programa factible para el siguiente problema:

TRABAJO	OPERACIÓN	TIEMPO EJECUCIÓN	MÁQUINA
A	1A	2	M1
	2A	3	M2
	3A	1	M3
B	1B	6	M3
	2B	5	M1
C	1C	4	M2
	2C	2	M3
	3C	3	M1
	4C	1	M2

Representación mediante diagrama de Gantt:



Es importante indicar que esta representación es para un programa concreto, pero hay infinidad de programas distintos que representan una solución válida del problema y por tanto infinidad de posibles representaciones.

Calcular la duración total del proceso o *makespan* mediante el diagrama de Gantt

Calcular el *makespan* mediante el diagrama de Gantt es muy sencillo, tan sólo hay que fijarse en la fecha de fin de la última actividad, en el ejemplo anterior el *makespan* sería 16, fecha en la que termina la última actividad, la 2B.

Determinar el camino crítico

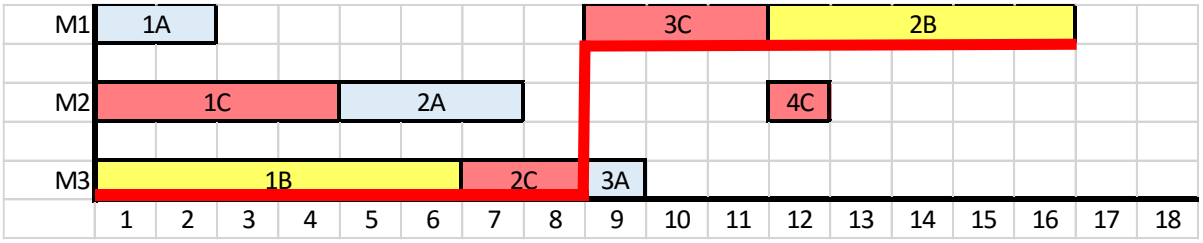
Entendemos por camino crítico, el conjunto de tareas sucesivas cuya suma de tiempos es la duración máxima del proyecto, y consecuentemente cualquier retraso en dichas actividades (que se denominan actividades críticas) implica un retraso en la duración total del proyecto [61].

De esta definición se pueden sacar dos conclusiones:

-Todo proyecto tiene, como mínimo, un camino crítico que, además, es el de mayor duración. Aunque también es posible que tenga varios caminos críticos.

-Las actividades críticas no pueden sufrir retrasos sin que a su vez, los sufra el proyecto, es decir tienen cero holgura o margen para ser ejecutadas.

Saber cuál es el camino crítico, permite saber cuáles son las actividades a las que hay que prestar mayor atención.

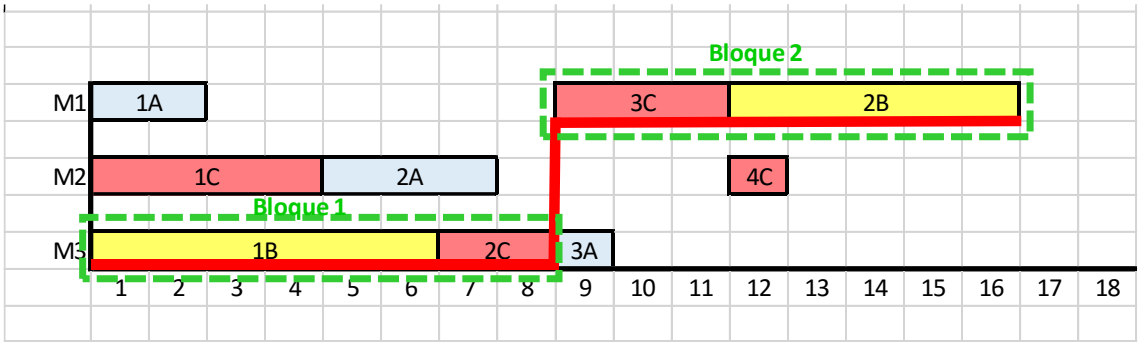


Camino crítico del ejemplo mostrado antes (representado por una línea roja). El camino crítico lo forman las actividades 1B-2C-3C-2B.

Un concepto importante relacionado con el camino critico son los llamados "bloques" que podemos definir como aparece en el artículo de R. Zhang et al. [76]:

Una secuencia de operaciones perteneciente al camino crítico se considera bloque si:

1. Contiene al menos dos operaciones.
2. Todas las operaciones de la secuencia pertenecen a la misma máquina.

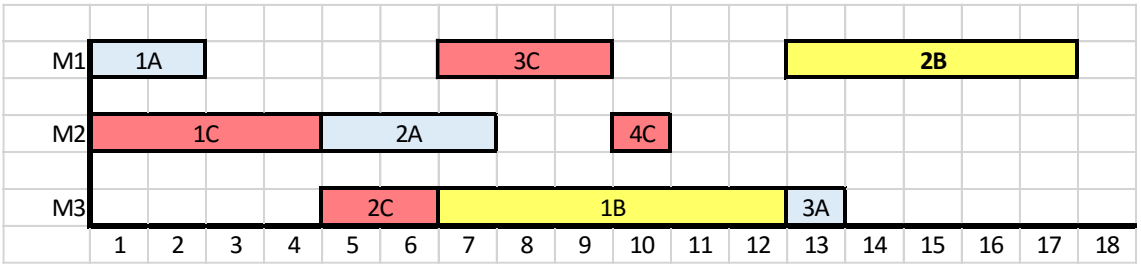


Bloques del ejemplo señalados con líneas verdes discontinuas. El Bloque 1 está compuesto por las operaciones 1B-2C y el Bloque 2 lo forman las operaciones 3C-2B.

Una forma de reducir la duración total es mediante el intercambio de actividades sucesivas pertenecientes a un mismo bloque, aunque esto no garantiza que siempre se reduzca, puede ocurrir o puede incrementar el *makespan.* A esta herramienta, la llamaremos *movimiento.*

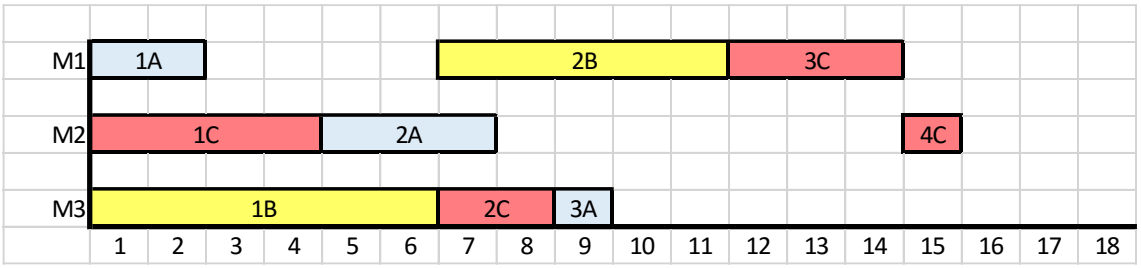
Respecto al ejemplo visto, se tiene dos posibles movimientos, uno en cada bloque:

Intercambio de 1B-2C (Bloque 1):



Este intercambio produce un aumento del *makespan*, antes valía 16 y ahora vale 17.

Intercambio 3C-2B (Bloque 2):



Sin embargo con este intercambio conseguimos mejorar el makespan, pasando de 16 a 15.

Es importante indicar que sólo cambia la secuencia de operaciones en la máquina en la que tiene lugar el intercambio. En las demás máquinas la secuencia es la misma, aunque el intercambio puede afectar a la fecha de comienzo de las actividades en esas otras máquinas.

1.6. Función objetivo.

En la definición del problema, se ha mencionado que lo que se busca es una solución que minimice el *makespan* o lo que es lo mismo, la duración total del proceso. Ésta, probablemente, sea la función objetivo más estudiada, pero no podemos obviar las demás.

Existen muchos objetivos diferentes a optimizar para el problema del *job shop*, a continuación se citarán los más importantes [9].

Las funciones objetivo están relacionadas con los tiempos de finalización de las operaciones C_{ij}, determinados por el programa de

producción. El tiempo de finalización de la última tarea, o lo que es lo mismo la fecha más tardía de finalización de los trabajos, C_j es lo que nos da el *makespan*.

Makespan (C_{max}): tiempo de finalización de la última operación.

$$C_{max} = \max_j\{C_j\}$$

Retraso máximo (T_{max}): consideramos el retraso de un trabajo j como la diferencia positiva entre la fecha de finalización (c_j) y la fecha en la que debería estar terminado (Due date: Dj). Si este valor es negativo, es decir el trabajo está terminado antes de la fecha acordada, el retraso vale cero, es decir $T_j = max\{0, c_j - Dj\}$. El objetivo es reducir la demora del trabajo que más se retrasa (retraso máximo).

$$T_{max} = \max_j\{T_j\}$$

Retraso medio (AT): esta función no pretende minimizar únicamente el retraso máximo, sino el retraso medio, que tiene en cuenta todos los trabajos.

$$AT = \frac{1}{N}\sum T_j$$

Tiempo flujo máximo (Fmax): llamamos tiempo de flujo o *flow time* (F_j) al tiempo en el que un trabajo está en el proceso, es decir, la diferencia de tiempo entre el momento que se termina de ejecutar y el momento desde el cual está disponible, F_j=C_j- r_j. Por tanto esta función pretende minimizar el máximo tiempo que un pedido está en el taller.

$$F_{max} = \max_j\{F_j\}$$

Tiempo Flujo Medio (AFT): esta función al igual que la anterior, trabaja con el tiempo de flujo, pero su objetivo es minimizar el tiempo de flujo medio de todos los trabajos del proceso.

$$AFT = \frac{1}{N}\sum F_j$$

Las que se acaban de citar son las más comunes, pero también podemos encontrar:

Número de Trabajos Retrasados (NTJ):

$$AU = \sum U_j$$

Suma pondeada de los retrasos (WT):

$$WT = \sum w_j T_j$$

Suma ponderada del tiempo de flujo (WFT):

$$WFT = \sum w_j F_j$$

Todas estas funciones objetivo se pretenden minimizar. Cada una de ellas es importante para un área y una circunstancia diferente, ya que el objetivo general es reducir el coste, y éste puede estar relacionado con cualquiera de las medidas de desempeño citadas anteriormente [22].

1.7. Métodos de resolución

Desde que se plantea el problema, en los años 50, han sido muchos los métodos que se han aplicado para su resolución. En el apartado de historia del JSSP se han mencionado algunos de ellos y los autores que los usaron para resolver el problema.

En general, estos métodos se pueden clasificar en dos categorías: métodos exactos y métodos aproximados. Los métodos exactos se basan en la enumeración de las soluciones del problema y usan un modelo matemático. En cambio, los métodos aproximados resuelven el problema encontrando una solución aproximada, que no tiene por qué ser la óptima. La herramienta que utilizan son algoritmos heurísticos y metaheurísticos aplicados de forma iterativa [58].

1.7.1. Métodos exactos.

Ramificación y poda (Branch and bound)

Para aplicar este método, se utiliza un árbol de soluciones, donde cada rama nos lleva a una posible solución, obtenida a partir de la actual. Para no tener que analizar cada una de las soluciones posibles, se utiliza la herramienta de "poda", en el momento en el que se detecta que una ramificación puede que no esté aportando soluciones óptimas. De esta forma se "poda" dicha rama y ya no se sigue ramificando en pro de las que están produciendo buenas soluciones.

Para cada subconjunto o rama, se calcula el valor de la mejor solución mediante un límite inferior. En el caso de que dicho límite inferior exceda el valor del mejor límite superior conocido, la rama puede ser descartada.

Dentro de la categoría de ramificación y poda, podemos encontrar algunas variantes, como la *ramificación y corte*, que combina planos de corte, ramificación y acotación. Los planos de corte mejoran la relajación del problema para acercarse con una mejor aproximación al problema de programación entera, y la ramificación y acotación, por su parte, continúa dividiendo el problema y acercándose a la solución final de éste [22].

1.7.2. Métodos de aproximación.

Reglas de Prioridad

Las reglas de prioridad se usan habitualmente, debido a su facilidad de implementación. El algoritmo de *Giffler y Thompson* es el más conocido dentro de este campo. Este algoritmo selecciona en cada paso la máquina que antes puede finalizar. De las operaciones que se realizan en esa máquina, se toman las que pueden comenzar antes de esa fecha más temprana de finalización [9].

Heurística de cuello de botella

Este método descompone el problema en tantos subproblemas de una sola máquina como máquinas tengamos, y la secuenciación de trabajos se hace iterativamente en el recurso más crítico. Entendemos por recurso crítico aquel que ralentiza todo el proceso de producción global, ya sea por ser el más lento o el que más trabajo asignado tiene. Para su aplicación se consideran todas las operaciones que se procesan en la máquina crítica y se secuencian según una regla de prioridad.

Reglas de prioridad más frecuentes [9]:

-Regla SDD (Shortest Due Date): secuenciar primero los trabajos más urgentes. Se minimiza el retraso máximo. Para estas funciones objetivo, es un algoritmo exacto.

-Regla SPT (Shortest Process Time): Secuenciar primero los trabajos de menor duración. Esta regla minimiza el tiempo de flujo medio (AFT). Para esta función, es un algoritmo exacto.

-Regla LPT (Longest process time): secuenciar primero los trabajos de mayor duración. Con esta regla se busca que el retraso, sea el mínimo posible. Para esta función, es un algoritmo exacto.

-Caso de tiempos de preparación dependientes de la secuencia: este es el caso en el que el tiempo de preparación de un trabajo depende del fabricado inmediatamente antes. El objetivo es minimizar el tiempo total de preparación, esto es un problema complejo. Una forma de resolverlo es mediante el Algoritmo de Kauffman (algoritmo heurístico constructivo).

Búsqueda local

Este tipo de algoritmos son más complejos y aparecen en los años 70. Se basan en una estructura de entorno, esto es un conjunto de soluciones que se generan a partir de una inicial. De esta forma nos movemos de solución en solución en el espacio de búsqueda, aplicando cambios locales, hasta que se cumpla un determinado criterio de parada. Este criterio de parada puede ser por ejemplo que se alcance la solución óptima o que se realice un número máximo de iteraciones.

Dentro de esta categoría, podemos encontrar entre otros, la búsqueda tabú (que desarrollaremos más adelante) y el recocido simulado (*simulated annealing* en inglés).

La búsqueda tabú se caracteriza por la premisa de intentar escapar de óptimos locales y continuar la búsqueda hacia soluciones aún mejores. Para ello almacena en una lista, llamada lista tabú, los movimientos más recientes, favoreciendo así moverse hacia soluciones no visitadas.

El Recocido Simulado está inspirado en el comportamiento de un sólido fundido a altas temperaturas, el cual se enfría lentamente para llegar al estado de mínima energía. Para su implementación se genera una sucesión de soluciones mediante una función. Y se elige una u otra en función de su probabilidad asociada. Esta probabilidad se denomina temperatura por su analogía con el concepto físico. Al igual que en la búsqueda tabú, esto se repite de forma iterativa hasta encontrar el óptimo o alcanzar un número máximo de iteraciones [57]. Mientras que la búsqueda tabú es un método determinista[1], el recocido simulado es probabilístico[2].

Algoritmos evolutivos y algoritmos de inteligencia de enjambre

Estos algoritmos aparecen en los años 80 y son una rama de la inteligencia artificial. Están inspirados en la naturaleza, en concreto en ciertos sistemas biológicos y en la base genético-molecular. Se parte de una población de individuos (soluciones), y se la hace evolucionar mediante unas modificaciones aleatorias, inspiradas en la evolución biológica, así como una selección en base a un criterio, el cual tiende a favorecer a los mejores individuos. Por tanto en cada iteración en vez de procesar una sola operación como en la búsqueda local, procesa un conjunto de ellas al mismo tiempo, llamado población.

Los más famosos de esta categoría son los algoritmos genéticos, los cuales imitan el proceso darwiniano de selección natural. La población inicial se genera aleatoriamente y en cada iteración va evolucionando gracias a tres

[1] Método determinista: las mismas entradas producen las mismas salidas, sin influencia del azar ni de la probabilidad.
[2] Método probabilístico: las salidas del sistema están determinadas tanto por acciones predecibles del proceso como por elementos aleatorios.

elementos clave: operador de selección, operador de cruce y operador de mutación. Hablaremos en detalle de este método más adelante.

Otro ejemplo de algoritmo de inteligencia artificial es la inteligencia de enjambre, inspirados en ciertos sistemas biológicos. Al igual que los algoritmos genéticos utilizan poblaciones, en la que los individuos de la población interactúan localmente entre ellos y con el medio ambiente. Los ejemplos más conocidos de algoritmos de inteligencia de enjambre son: ACO (Algoritmo colonia de hormigas), enjambre de partículas, ABC (algoritmo de colonia artificial de abejas), etc.

El algoritmo de colonia de abejas se desarrollará en capítulos posteriores.

Capítulo 2

Algoritmos Genéticos.

2.1. Introducción.

En este capítulo, se explicará con más detalle los antes mencionados, algoritmos genéticos (AG). Este método de búsqueda es una poderosa herramienta inspirada en la teoría de la evolución de Darwin (1859), en la que los individuos con unas características concretas (los más adaptados al medio) tienen más posibilidad de sobrevivir, y además traspasan su herencia genética a su descendencia.

Los algoritmos genéticos son métodos adaptativos no determinísticos, y buscan en el espacio de soluciones para obtener el óptimo, o al menos una solución lo más cerca posible.

Los algoritmos genéticos forman parte de los llamados algoritmos evolutivos, que surgieron a principios de los años 60, aunque cobraron especial relevancia cuando J. Holland [39] en 1975 incorporó las características de selección y supervivencia natural a la resolución de problemas de Inteligencia Artificial [46]. Desde esa fecha han aparecido numerosos trabajos sobre algoritmos evolutivos y sus diferentes variantes, como las estrategias evolutivas, los algoritmos genéticos y la programación genética.

Aunque sin duda, han sido los AG los que más se han utilizado, tanto por los buenos resultados que aportan como por su adaptación a las condiciones reales de trabajo, además de su fácil implementación computacional [55]. Otro punto a favor de ellos es su robustez, porque independientemente de la solución inicial, siempre dan resultados de buena calidad.

Esta técnica se basa en poblaciones, lo cual quiere decir que no analiza de una en una las soluciones (como por ejemplo hace la búsqueda tabú), sino que utiliza conjuntos de soluciones (población) de forma simultánea durante la búsqueda.

Esta población inicial se genera de forma aleatoria. Después, en cada iteración la población se somete a tres elementos clave (operadores genéticos): selección, cruce y mutación. Así, el mejor individuo de la población tiene más oportunidades de sobrevivir (ser seleccionado), y por lo tanto también de reproducirse. En el cruce se produce un intercambio de material genético entre ambos progenitores. Asimismo, el operador de mutación introduce diversidad en la búsqueda, permitiendo acercarnos a otras zonas aún no visitadas del espacio de soluciones.

De esta manera, mediante los AG y sus operadores genéticos se consigue llevar la búsqueda en cada iteración hacia soluciones más favorables, al igual que sucede en la naturaleza gracias a la selección natural.

2.2. Paralelismo con la Biología.

Como ya hemos dicho, los algoritmos genéticos, y en general los algoritmos evolutivos están inspirados en la teoría de la evolución, planteada por C. Darwin en su obra “El origen de las especies” en el año 1859. En ella postuló que todos los seres orgánicos descienden de un mismo antepasado común, evolucionando hasta lo que son hoy mediante el proceso de selección natural.

La selección natural actúa paulatinamente, los individuos van poco a poco acumulando características genéticas ventajosas para el medio en el que viven, y de esta forma, los individuos más desfavorecidos van desapareciendo en pro de los que están mejor adaptados, que además se reproducirán más fácilmente y su descendencia heredará dichas cualidades que los hacen mejores [68]. Siendo la consecuencia final, un cambio adaptativo en la población.

Otro aspecto a tener en cuenta es la *mutación*, gracias a este elemento, la naturaleza induce cambios aleatorios en los individuos, que pueden resultar ventajosos o no. Si dicho cambio favorece la adaptación del individuo, tenderá a mantenerse en futuras generaciones; mientras que si el cambio que introduce la mutación es desfavorable, lo más probable es que dicha característica desaparezca.

Los algoritmos genéticos beben directamente de la evolución biológica, por lo que a continuación veremos los conceptos clave que usan los AG y su término análogo en la biología:

Término biológico	Término en AG
Individuo (ser vivo): ente natural formado por cromosomas	**Solución:** elemento básico del algoritmo.
Población: conjunto de individuos de la misma especie que coexisten en un mismo lugar y tiempo y comparten unas mismas características biológicas	**Población:** subconjunto de soluciones factibles del problema
Genotipo: conjunto de factores heredados en un individuo	**Genotipo**: solución codificada del problema a resolver
Fenotipo: caracteres visibles que el individuo presenta como resultado de la interacción entre su genotipo y el medio	**Fenotipo:** solución decodificada y que adquiere significado cuando se utiliza con los datos del problema
Cromosoma: estructura que transporta los factores hereditarios o genes	**Cromosoma:** cadena de caracteres donde se almacena la solución codificada
Gen: segmento más corto en el que el cromosoma puede dividirse. Unidad de recombinación	**Gen:** número real o binario cuya unión forma el cromosoma
Loci: cada una de las posiciones que puede ocupar un gen dentro del cromosoma	**Loci:** cada una de las posiciones en la cadena de codificación.
Generación: cada uno de los reemplazos que se origina cuando se tiene descendientes	**Iteración**: cada uno de los pasos en los que se divide la búsqueda.

Paralelismos entre los términos biológicos y los algoritmos evolutivos. [55]

La biología trabaja con cromosomas (formados por genes) y en la reproducción, dos individuos intercambian entre sí segmentos de los cromosomas (genes), dando lugar a un nuevo individuo que posee características de ambos progenitores.

Esto, traducido a la estructura de algoritmos genéticos, significa que dos soluciones representadas cada una por una cadena de caracteres se combinan entre sí (intercambiando algunos de los números de la cadena), obteniendo una nueva cadena o solución.

2.3. Codificación de las soluciones.

Una de las decisiones que hay que tomar primero a la hora de aplicar un algoritmo genético a la resolución de un problema concreto es qué regla de codificación se va a utilizar.

Para que una codificación resulte óptima, debe cumplir:

-Todas las soluciones del problema se deben poder representar mediante la codificación.

-Todas las soluciones codificadas deben dar lugar a una solución válida del problema.

A continuación se explicarán las normas de codificación más comunes [55], para después analizar cuál es la que mejor puede representar nuestro problema de estudio, el *job shop*.

Codificación binaria

Es la más extendida, dado que es la que se usó en los primeros algoritmos genéticos y por tanto la mayoría de obras posteriores se sustentó en dicha codificación.Como su nombre indica, cada solución se representa como una cadena de bits que pueden ser 0 ó 1.

Un ejemplo sobre el que se aplica esta codificación es el *problema de la mochila*. En este problema, tenemos una serie de objetos que se quieren meter en una mochila de capacidad limitada. El objetivo es, sin superar dicha capacidad, obtener el máximo beneficio (cada objeto tiene asignado un beneficio y un peso) [10]. Una posible representación de una solución a este problema sería una cadena de caracteres en la que el número 1 represente si metemos el objeto en la mochila, y el número 0 si no lo hacemos.

Esta codificación tiene dificultades a la hora de adaptarse a muchos otros problemas, pues el esfuerzo necesario para ello complica en gran medida su implementación.

Codificación permutacional

Este tipo de codificación presenta cadenas de números enteros diferentes entre sí (no se repiten) y se utiliza sobre todo en problemas donde se necesita dar un orden.

El problema clásico que trabaja con esta codificación es el *problema del viajante,* también conocido como TSP (del inglés, *Travelling Salesman Problem*) en el que una persona debe visitar una serie de ciudades, tomando como punto de partida y punto final, la misma ciudad. El objetivo es encontrar la secuencia de ciudades que pasando por todas y cada una de ellas una sola vez, tenga el mínimo recorrido.Para codificar este problema, se asigna un número a cada ciudad; y una solución representa el orden por el que se visitan las ciudades.

Ejemplo: tenemos cinco ciudades y le asignamos un número a cada una de ellas. Una posible solución sería: 2 3 5 1 4. Esto significa que primero se visita la ciudad 2, después la ciudad 3 y así sucesivamente hasta visitar la ciudad 4 en último lugar.

Otro tipo de problema para el que esta codificación se ajusta bien es para la programación de operaciones en un taller tipo *flow shop permutacional*. En este caso, se tienen varias máquinas en serie y un conjunto de trabajos que tienen la misma ruta (orden por el que pasan por las máquinas). Además todas las máquinas procesan los pedidos en el mismo orden. El objetivo es decidir el orden de entrada de los trabajos a la cadena de máquinas.

Codificación permutacional con repetición

Esta codificación (al igual que la anterior) utiliza cadenas de números enteros, la diferencia es que aquí éstos números se repiten.

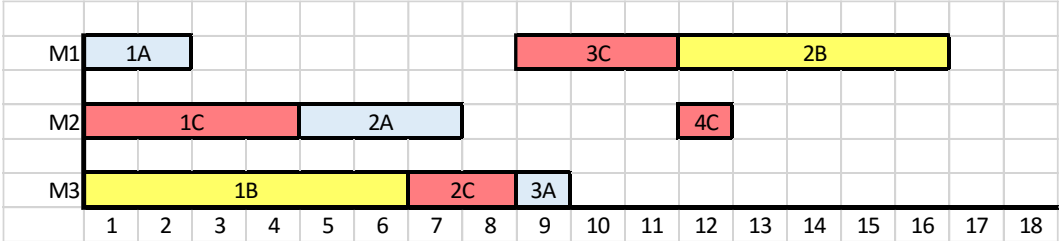
La codificación permutacional con repetición es la que mejor se adapta a nuestro problema de estudio, el *job shop*. A cada trabajo le es asignado un número entero, y dicho número se repite en la cadena de codificación tantas veces como operaciones tenga ese trabajo.

Para ilustrar esto, vayamos al ejemplo visto en el apartado de descripción del *job shop*:

TRABAJO	OPERACIÓN	TIEMPO EJECUCIÓN	MÁQUINA
A	1A	2	M1
	2A	3	M2
	3A	1	M3
B	1B	6	M3
	2B	5	M1
C	1C	4	M2
	2C	2	M3
	3C	3	M1
	4C	1	M2

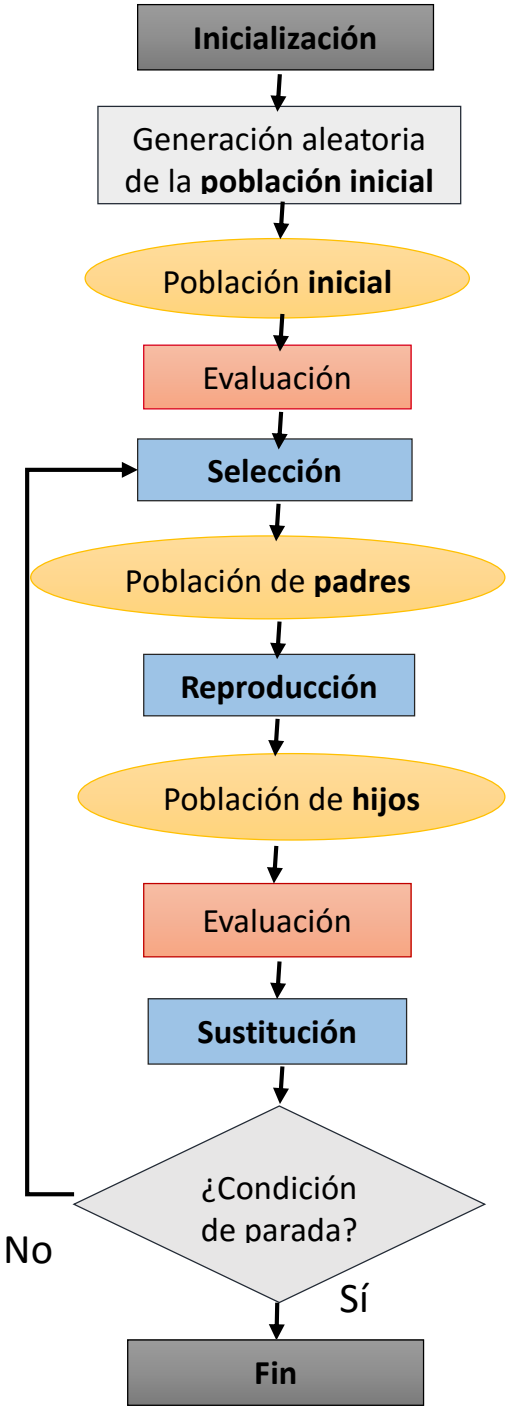
Tenemos 3 trabajos: A, B y C, que los asignaremos los números 1, 2 y 3 respectivamente. Una posible solución del problema sería:

Que correspondería al siguiente programa expresado mediante el diagrama de Gantt:



2.3. Estructura de los algoritmos genéticos.

La siguiente imagen muestra el esquema general del funcionamiento de los algoritmos genéticos. En los siguientes apartados, se verá detenidamente cada paso.



2.3.1. Inicialización.

Antes de nada, el primer paso es definir:

Norma de codificación: vistas en el apartado anterior.

Tipos de selección, cruce y mutación: se explicarán en sus respectivos apartados.

Tamaño de población: hay que tener en cuenta que cuanto más grande sea la población, mayor será el espacio de búsqueda explorado, pero la velocidad de procesamiento será más pequeña. Por lo que a la hora de decidir el tamaño de la población hay que buscar el equilibrio entre ambas características.

Probabilidad de cruce y de mutación: indican respectivamente la frecuencia con la que los padres se cruzan entre sí o mutan. Si la probabilidad de cruce es nula, los hijos son copias exactas de los padres, en cambio si es del 100%, los hijos siempre se obtienen por cruce. Si la probabilidad de mutación es nula, no se introduce nada de variabilidad en el algoritmo, en cambio, si es del 100%, todos los padres mutarían, lo que es contraproducente, pues la calidad de las soluciones degeneraría [10].

Criterio de parada: dependiendo del criterio de parada, será necesario elegir algunos parámetros como por ejemplo el número máximo de iteraciones o el máximo número de iteraciones en las que no se mejora el valor de la mejor solución. Esto se exlicará con más detalle en el apartado de condición de parada.

2.3.2. Generación aleatoria de la población inicial.

Una vez, se han decidido todos los parámetros del sistema, hay que obtener la población inicial.

La forma más habitual, es la generación aleatoria, esto garantiza que la población inicial esté uniformemente distribuida por el espacio de búsqueda. Esto lleva a la exploración de soluciones, aunque algunas de ellas puede que no sean muy buenas (de hecho la calidad de las soluciones de la población inicial suele ser baja), pueden llevarnos a óptimos, que quizá desde unas soluciones de mayor calidad, no podríamos haber accedido.

La generación mediante un método heurístico es otra forma de crear la población inicial, aunque es bastante menos frecuente. Tiene el problema mencionado anteriormente, que esas soluciones presumiblemente mejores, pueden no llevar al óptimo, esto es debido a la prematura convergencia del algoritmo (probablemente hacia óptimos locales).

El resultado de este paso del algoritmo es la población inicial.

2.3.3. Evaluación.

Cuando se evalúa una solución, lo que se está determinando es su calidad, que bien puede ser el valor de la función objetivo o el valor de una función de adecuación relacionada con la función objetivo.

Por ejemplo si estamos con el problema del viajante de comercio, el valor de la función objetivo será la longitud total del recorrido para un orden concreto de ciudades.

En el problema del *job shop*, ya hemos hablado anteriormente de las diferentes funciones objetivo. Un ejemplo sería el valor del *makespan*.

El operador evaluación actúa al comienzo del algoritmo para evaluar la población inicial, y además actúa en cada iteración evaluando la calidad de la población de hijos.

2.3.4. Selección.

No todos los individuos de la población inicial van a formar parte de la población de padres. Bajo la premisa de que los individuos mejor adaptados tienen más probabilidad de reproducirse, los individuos de la población inicial se seleccionarán para formar parte de la población de padres dependiendo de su calidad.

Un concepto importante es la *presión selectiva*, que indica en qué grado se favorece a las mejores soluciones. Cuanto mayor sea la presión selectiva, las soluciones mejores tendrán más posibilidades de ser elegidas y al revés en caso contrario. El hecho de tener una alta presión selectiva, nos lleva a mejores soluciones rápidamente, pero esto conlleva el riesgo de terminar en un óptimo local, además de perder diversidad [55]. Al igual que cuando se decide el tamaño de la población, también es necesario buscar un equilibrio.

A continuación se verán las principales técnicas de selección:

Método de la ruleta (selección proporcional)

Los individuos (soluciones) de más calidad, tienen más probabilidad de ser elegidos. Dicha probabilidad viene dada por la siguiente expresión:

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$$

Siendo p_i la probabilidad de la solución *i* de ser elegida, f_i el valor de la función de aptitud[3] de la solución *i*, y en el denominador encontramos la suma de los valores de la función de aptitud, para las *n* soluciones que hay en total[4].

Por tanto, la probabilidad de que una solución sea elegida está relacionada directamente con el valor de la función de aptitud. Este valor no tiene por qué ser el mismo que el de la función objetivo, aunque están estrechamente relacionados.

Esto es así, debido a que por ejemplo para el caso del problema del *job shop*, en el que queremos reducir el *makespan*, una solución tendrá más calidad cuanto más pequeño sea el valor de la función objetivo (*makespan* en este caso). Por tanto si la expresión anterior utilizara directamente la función objetivo, obtendríamos que una solución buena (*makespan* pequeño), tendría menos posibilidades de ser elegida.

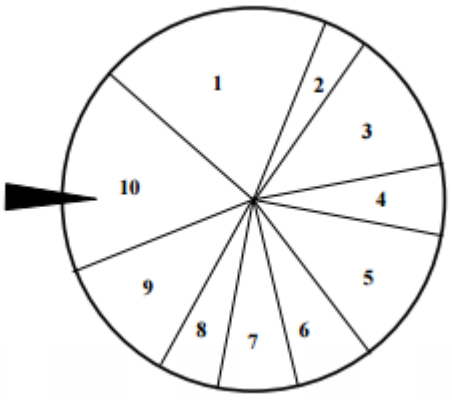
Para solucionar esto, se introduce la función de aptitud, que para el ejemplo visto, viene dada por:

$$f_i = \frac{1}{C_i},$$

siendo C_i el valor de la función objetivo para la función *i.*

De esta forma, se consigue que los mínimos valores del *makespan*, sean los que tienen más posibilidades de salir. Además de para el ejemplo visto, también se utiliza para cualquier función objetivo que haya que minimizar. Si por el contrario, se quiere maximizar, en ese caso tendremos que: $f_{i=C_i}$.

Una vez calculada la probabilidad de cada solución, lo representamos mediante una ruleta dividida en n porciones (una por cada solución). El tamaño de la porción será proporcional a la probabilidad calculada para esa solución.

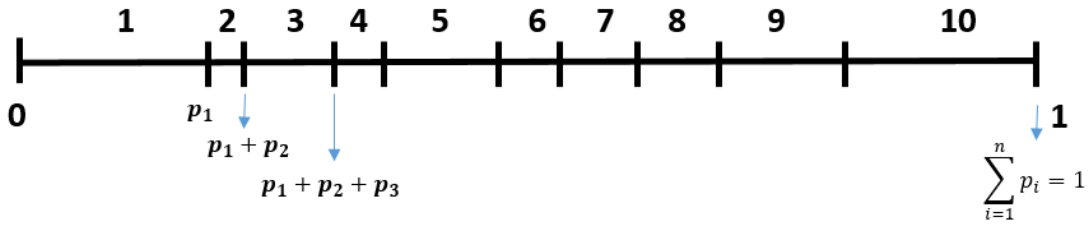


Método de selección de la Ruleta. [55]

[3] Función de aptitud: *fitness function* (en inglés).
[4] El tamaño de la población es *n*.

La selección se lleva a cabo girando *n* veces la ruleta, y la sección en la que caiga el marcador indicará el individuo que pasa a la población de padres.

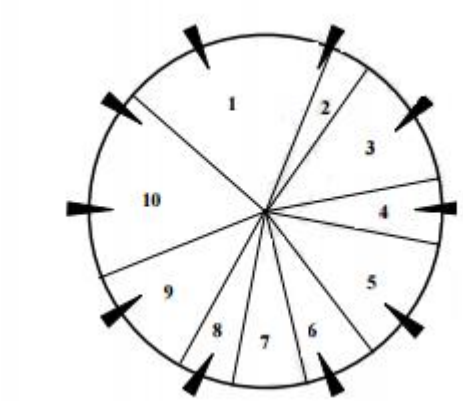
En la práctica se hace una representación lineal, y al igual que en la ruleta, cada trozo de línea representa una solución y su tamaño viene dado por la probabilidad.



Girar la ruleta equivale a dar un número al azar entre 0 y 1, y se selecciona la sección a la que pertenezca ese número.

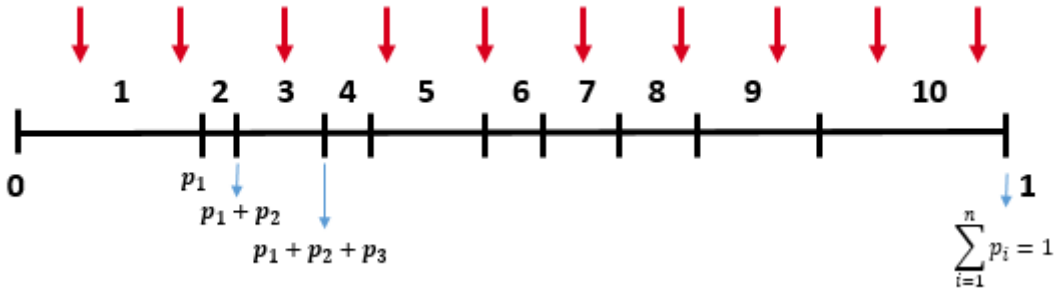
El inconveniente de esta técnica, es que se favorece demasiado a los súperindividuos (los individuos de mayor calidad), dado que además las soluciones se pueden repetir. Esto lleva a una rápida convergencia debido a la alta presión selectiva. Para evitar este problema, se introduce una variación en el método de la ruleta: el llamado **método de selección estocástica universal.**

Este método es muy similar al método de la ruleta, funciona de igual forma asignando una porción de la ruleta según su probabilidad, sólo que en lugar de tener un único marcador, se tienen *n* marcadores distribuidos a la misma distancia. De esta forma, en una sola tirada se escogen los *n* individuos de la población de padres, sin necesidad de generar números aleatorios.



Método de selección estocástico universal.

La representación lineal de este método sería:



En la que las flechas rojas representan los marcadores de la ruleta, por tanto las soluciones seleccionadas serían: 1, 1, 3, 5, 5, 7, 8, 9, 10, 10.

Al utilizar esta variante del método de la ruleta, se consigue aumentar la diversidad de las soluciones que pasan a formar parte de la población de padres, o lo que es lo mismo, relajar la presión selectiva.

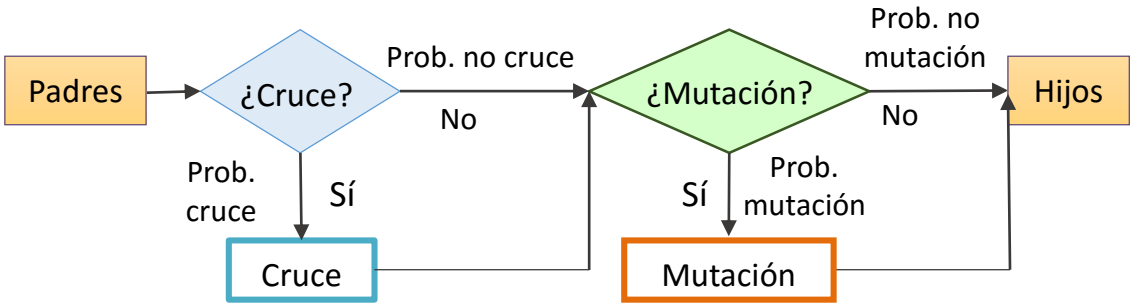
Selección por torneo.

Se escogen de forma aleatoria dos individuos para competir entre sí. El que tenga mayor calidad, será el seleccionado para formar parte de la población de padres. Este paso se repite *n* veces, hasta obtener *n* padres. Como se puede deducir, en la población de padres, un mismo individuo puede aparecer varias veces (al igual que en el método de la ruleta y en el estocástico universal).

Normalmente son dos los individuos que se enfrentan entre sí, aunque pueden ser más. Cuanto mayor sea el número de individuos a competir, mayor es la presión selectiva que se ejerce.

2.3.5. Reproducción.

La reproducción consta de dos partes diferenciadas en las que actúan los operadores cruce y mutación respectivamente. El esquema que se sigue es el siguiente:



El primer paso es agrupar la población de padres en parejas. Cada pareja de padres se somete simultáneamente al proceso de reproducción. Para ello, lo primero es ver si se van a cruzar o no, esto depende de la probabilidad de cruce. Se genera un número aleatorio del 0 al 1 y si es menor que el número que indica la probabilidad, se produce el cruce. Ejemplo: si la probabilidad de que una pareja de padres se cruce es del 80%, y el número aleatorio está entre 0 y 0.8, los dos padres se cruzan, pero si por el contrario, el número aleatorio es mayor que 0.8, pasaría directamente al siguiente paso: la mutación. El operador mutación actúa de la misma forma que acabamos de describir, se da un número aleatorio y si entra dentro de la probabilidad de mutación, los dos individuos mutan y en caso opuesto, no. Con el objetivo de no causar mucha desviación en el algoritmo, los valores de la probabilidad de mutación suelen ser pequeños (entre el 0.1 y el 5%) [63].

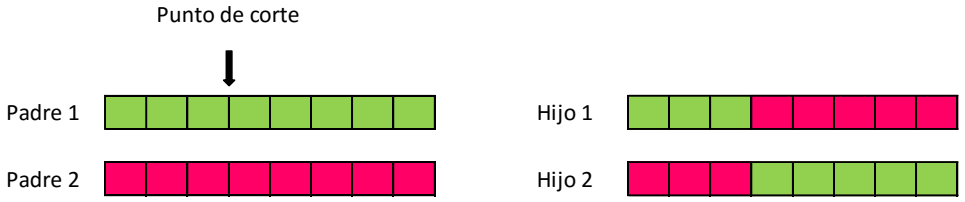
En el cruce, los hijos son el resultado del intercambio de material genético entre los dos padres, en la mutación, los hijos son el resultado de un cambio en la cadena de un solo padre (sin intervención del otro).

A continuación se detallará algunos ejemplos de clases de cruce y de mutación que existen.

2.4.5.1. Cruce.

Operador con un punto de cruce

Es la forma más simple del operador cruce. Consiste en elegir aleatoriamente la posición por la que se va a efectuar el cruce e intercambiar las partes de los padres divididas por dicha posición:



Operador con varios puntos de cruce

Es una variante del modelo anterior. Se generan de forma aleatoria tantas posiciones de la cadena como puntos de corte haya y al igual que antes, se intercambian las partes de la cadena entre dichos puntos.
Ejemplo para tres puntos de corte:

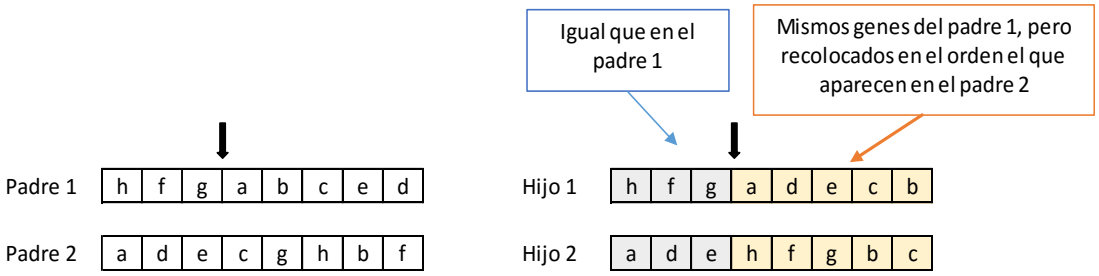
Operador de cruce uniforme

Cada gen de la cadena de los descendientes se puede obtener de forma aleatoria de la cadena de los padres. Para ello se utiliza una máscara, cadena formada por ceros y unos. Dependiendo si hay un cero o un uno, el hijo toma ese gen de un padre o del otro:

Operador de cruce Ordenado (OX)

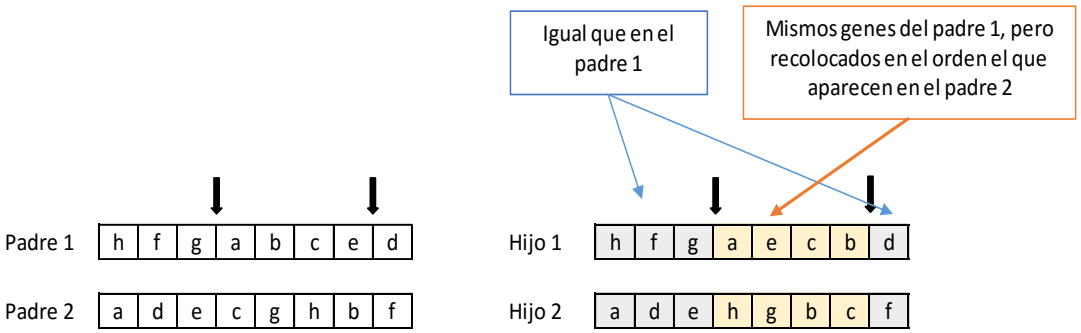
Este operador está especialmente desarrollado para la codificación permutacional. Esto se debe a que si se aplica los operadores previamente explicados (utilizados para codificaciones binarias) a dicha codificación, la cadena resultante podría no ser válida.

El operador de cruce ordenado funciona de la misma forma que los operadores con un punto de cruce, pero la diferencia es que en lugar de coger en trozo de cadena directamente del otro progenitor, éste sólo se utiliza para determinar el orden en el que se van a recolocar los genes [63]:



Operador de cruce ordenado con varios puntos

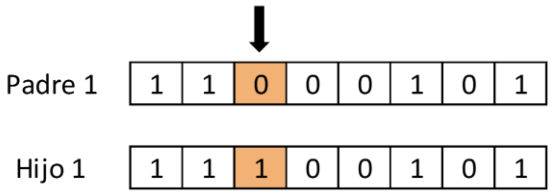
Funciona de la misma manera que el operador de cruce ordenado, pero dando varios puntos de cruce:



2.4.5.2. Mutación.

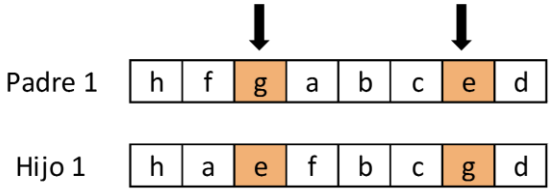
Mutación estándar

Operador por excelencia para el tipo de codificación binaria. Cada gen tiene una probabilidad distinta de ser mutado (esta probabilidad es menor del 1%) [55]. Mediante números aleatorios y según dicha probabilidad, se deciden los genes que mutan (pasan de 0 a 1 y al revés):



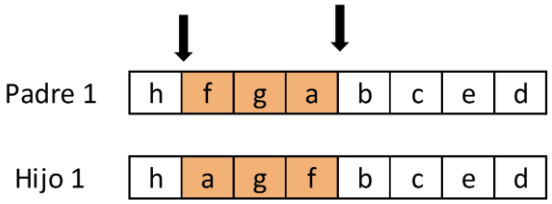
Mutación basada en el orden

Este operador intercambia los genes de dos posiciones obtenidas al azar:



Mutación por Inversión

Dos puntos de la cadena son seleccionados aleatoriamente, y el trozo de cadena situado entre dichos puntos invierte su orden:



2.3.6. Sustitución elitista.

Una vez obtenidos los hijos y evaluados, se aplica el operador de sustitución, en concreto la sustitución elitista. Para ello, se selecciona el peor de los hijos y se sustituye por el mejor de la población inicial de la iteración actual. Es decir, la población inicial de la siguiente iteración serán todos hijos menos el peor de ellos, más el mejor individuo de la población inicial. De esta

forma, garantizamos que la mejor solución conocida permanezca en el proceso [55].

2.3.7. Condición de parada.

Las condiciones de parada pueden ser varias. Una de ellas es un número límite de iteraciones, si dicho límite se alcanza nos quedamos con la mejor solución encontrada hasta ese momento. Otro criterio de parada es si la mejor solución hasta el momento no se mejora en un número dado de iteraciones, en este caso al igual que antes nos quedaríamos con la mejor solución encontrada hasta el momento.

Con que se cumpla un solo criterio de parada, el algoritmo se detendría y llegaría a su fin. Si no se cumple ningún criterio de parada, el algoritmo pasa a la siguiente iteración, pero esta vez comenzando desde el paso 4, el paso de selección. Esto se debe a que ya no es necesario inicializar el algoritmo, ni crear y evaluar la población inicial, puesto que la población inicial de la nueva iteración está formada por la población de hijos de la anterior iteración.

2.4. Ejemplo resuelto.

Enunciado del problema: 4 trabajos x 4 máquinas. Calcular el mínimo *makespan*.

Trabajo	A				B				C				D			
Operación	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Tiempo ejecución (p_i)	3	5	9	6	1	9	2	5	1	1	5	2	4	2	6	1
Máquina (M_i)	1	2	4	3	2	1	3	4	4	1	3	2	3	2	1	4

Procedimiento del algoritmo:

Paso 0: Inicialización.

* Codificación: Permutacional con repetición
* Selección: método de la Ruleta.
* Cruce: *Order crossover* (OX)
* Mutación: mutación basada en el orden.
* Parámetros:
  - Tamaño población: 4
  - Probabilidad de cruce: 80%
  - Probabilidad de mutación: 10%
  - Criterio de parada: máximo 4 iteraciones.

1º Iteración

Paso 1: Generación aleatoria.

Tamaño población = número de soluciones que hay que generar=4.

Generamos cuatro soluciones aleatorias:

ABCADBDCACBBACDD

BBCCAADDCCAADDBB

CDABBADCAADCBBCD

CDACBADCBAABCDDB

Paso 2: Evaluación

X_1) ABCADBDCACBBACDD

$$Cmax = 26$$

X_2) BBCCAADDCCAADDBB

$$Cmax = 37$$

X_3) CDABBADCAADCBBCD

$$Cmax = 36$$

X_4) CDACBADCBAABCDDB

$$Cmax = 30$$

Paso 3: Selección.

A partir de la población inicial obtenemos la población de padres. Si partíamos, en este caso, de cuatro individuos, la población de padres será también de cuatro individuos.

Cada individuo tiene un valor, y en base a ese valor se calcula su probabilidad de salir. Se dan cuatro números (porque es el número de población) al azar y se eligen cuatro padres. Éstos pueden repetirse.

P_1= 0'304 → 30,4%

P_2= 0'213 → 21,3%

P_3= 0'219 → 21,9%

P_4= 0'264 → 26,4%

Pasamos ahora a dar cuatro números al azar entre 0 y 1, y mediante el método de la ruleta las soluciones elegidas en base a esos números son las que aparecen entre paréntesis:

0'57 (X_3) 0'21 (X_1) 0'80 (X_4) 0'93 (X_4)

Paso 4: Reproducción (cruce y mutación).

Juntamos a los padres de dos en dos y hacemos que se crucen o muten según la probabilidad.

Pareja X_3 – X_1

-Cruce: Primero damos un número al azar entre 1 y 100, si el número está entre 0-20 → no se cruzan, si está entre 20-100 → sí cruzan.

Número al azar: 66 → Sí cruzan.

X_3: CDABBADCAADCBBCD → CDABBAACADDCBBCD

X_1: ABCADBDCACBBACDD → ABCADBCDABBCACDD

-Mutación: Ahora damos un número al azar entre 1 y 100 para ver si mutan. Si el número está entre 0-90 → no mutan; si está entre 90-100 → sí mutan.

Número al azar: 49 → No mutan.

Luego del cruce de la pareja X_3 – X_1 obtenemos los siguientes Hijos:

CDABBAACADDCBBCD (H_1)

ABCADBCDABBCACDD (H_2)

Pareja X_4 – X_4

-Cruce: Número al azar: 8 → No cruzan.

-Mutación: Número aleatorio para ver si mutan: 93 → Sí mutan.

Elegimos 2 posiciones al azar (hijos) y las intercambiamos.

X_4: CDACBADCBAABCDDB → CDACBADCBAABCDBD

X_4: CDACBADCBAABCDDB → CDACBADCBAABCBDD

Luego del cruce de la pareja X_4 – X_4 obtenemos los siguientes Hijos:

CDACBADCBAABCDBD (H_3)

CDACBADCBAABCBDD (H_4)

Paso 5: Evaluación hijos.

Evaluamos los nuevos individuos (los hijos).

1º) H_1
CDABBAACADDCBBCD → $Cmax = 36$

2º) H_2
ABCADBCDABBCACDD → $Cmax = 25$

3º) H_3
CDACBADCBAABCDBC → $Cmax = 31$

4º) H_4
CDACBADCBAADCDBC → $Cmax = 31$

Paso 6: Sustitución elitista.

Una vez tenemos los hijos, seleccionaremos todos menos el peor y los pasaremos a la siguiente población (ésta será la población inicial en la siguiente iteración). El hueco que quede se rellena con el mejor individuo de la población inicial (con la que partíamos al comienzo de esta iteración, compuesta de X_1, X_2, X_3 y X_4).

Hijos: H_1 $Cmax = 36$ → Éste individuo se queda fuera, es la peor solución.

H_2 $Cmax = 25$

H_3 $Cmax = 31$

H_4 $Cmax = 31$

Mejor individuo de la población inicial: X_1 con $Cmax = 26$ → éste como mejor individuo de la población inicial, sustituye a H_1.

Paso 7: ¿Condición de parada?

No se cumple.

2º Iteración

En ésta y en futuras iteraciones los pasos 1 (generación aleatoria) y 2 (evaluación) ya no son necesarios. Esto es, porque ya tenemos formada la población inicial de la iteración anterior (compuesta por: X_1, H_2, H_3 y H_4; a quien denominaremos como: X_1, X_2, X_3 y X_4 para evitar confusiones con los hijos (H_n) que obtengamos en esta iteración.

X_1 = ABCADBDCACBBACDD. $Cmax = 26$

X_2 = ABCADBCDABBCACDD. $Cmax = 25$

X_3 = CDACBADCBAABCDBC. $Cmax = 31$

X_4 = CDACBADCBAADCDBC. $Cmax = 31$

Así, pasaremos directamente al paso 3.

Paso 3: Selección.

Probabilidades de selección:

P_{x1}= 0'269 → 26,9% P_{x2}= 0'279 → 27,9%

P_{x3}= 0'226 → 22,6% P_{x4}= 0'226 → 22,6%

Pasamos ahora a dar cuatro números al azar entre 0 y 1 y elegimos las soluciones con el método de la ruleta:

0'85 (X_4) 0'37 (X_2) 0'19 (X_1) 0'73 (X_3)

Paso 4: Reproducción (cruce y mutación).

Pareja X_4 – X_2

-¿Cruzan? Número al azar: 98 → Sí cruzan.

X_4: CDACBADCBAABCBDD → CDABCADCBAABCBDD (H_1)

X_2: ABCADBCDABBCACDD → ABCDABCDABBCACDD (H_2)

-¿Mutan? Número al azar: 55 → No mutan.

Pareja X_1 – X_3

-¿Cruzan? Número al azar: 9 → No cruzan.

-¿Mutan? Número al azar: 39 → No mutan.

Luego del cruce de la pareja X_1 – X_3 no obtenemos hijos.

Paso 5: Evaluación hijos.

Evaluamos los nuevos individuos (los hijos).

1º) H_1

CDABCADCBAABCBDD → $Cmax = 31$

2º) H_2

ABCDABCDABBCACDD → $Cmax = 25$

3º) $H_3 = X_1$ (Esto es, como de X_1 y X_3 no obtenemos hijos, mantenemos a los padres en la nueva población).

ABCADBDCACBBACDD → $Cmax = 26$

4º) $H_4 = X_3$

CDACBADCBAABCDBC → $Cmax = 31$

Paso 6: Sustitución elitista.

Pasamos todos los hijos menos el peor a la siguiente población.

Hijos: H_1 $Cmax = 31$ → Éste individuo se queda fuera, es peor solución.

H_2 $Cmax = 25$

H_3 $Cmax = 26$

H_4 $Cmax = 31$

Mejor individuo de la población inicial: X_2 con $Cmax = 25$ → éste como mejor individuo de la población inicial, sustituye a H_1.

Paso 7: ¿Condición de parada?

No se cumple.

3º Iteración

Población inicial:

X_1 = ABCADBCDABBCACDD. $Cmax = 25$

X_2 = ABCDABCDABBCACDD. $Cmax = 25$

X_3 = ABCADBDCACBBACDD. $Cmax = 26$

X_4 = CDDCBADCBAABCDBC. $Cmax = 31$

Paso 3: Selección.

Probabilidad de selección:

P_{x1}= 0’265 → 26,5%

P_{x2}= 0’265 → 26,5%

P_{x3}= 0’255 → 25,5%

P_{x4}= 0’215 → 21,5%

Pasamos ahora a dar cuatro números al azar entre 0 y 1 y mediante el método de la ruleta se eligen las soluciones:

0’39 (X_2) 0’68 (X_3) 0’77 (X_3) 0’95 (X_4)

Paso 4: Reproducción (cruce y mutación).

Pareja X_2 – X_3

-¿Cruzan? Número al azar: 42 → Sí cruzan.

X_2: ABCDABBCDABBCACDD → ABCDAABCDBBCACDD (H_1)

X_3: ABCADBDCACBBACDD → ABCADABCDBCBACDD (H_2)

-¿Mutan? Número al azar: 31 → No mutan.

Pareja X_3 – X_4

-¿Cruzan? Número al azar: 17 → No cruzan. (H_3=X_3)
-¿Mutan? Número al azar: 25 → No mutan. (H_4=X_4)

Paso 5: Evaluación hijos.

Evaluamos los nuevos individuos (los hijos).

1º) H_1
ABCDAABCDBBCACDD → $Cmax = 25$

2º) H_2
ABCADABCDBCBACDD → $Cmax = 25$

3º) H_3=X_3 (Esto es, como de X_3 y X_4 no obtenemos hijos, mantenemos a los padres en la nueva población).

ABCADBDCACBBACDD → $Cmax = 26$

4º) H_4=X_4

CDDCBADCBAABCDBC → $Cmax = 31$

Paso 6: Sustitución elitista.

Pasamos todos los hijos menos el peor a la siguiente población.

Hijos: H_1 $Cmax = 25$

H_2 $Cmax = 25$

H_3 $Cmax = 26$

H_4 $Cmax = 31$ → Éste individuo se queda fuera.

Mejor individuo de la población inicial: X_1 con $Cmax = 25$ → éste como mejor individuo de la población inicial, sustituye a H_4.

Paso 7: ¿Condición de parada?

No se cumple.

4º Iteración

Población inicial:

X_1 = ABCDAABCDBBCACDD. $Cmax = 25$

X_2 = ABCADABCDBCBACDD. $Cmax = 25$

X_3 = ABCADBDCACBBACDD. $Cmax = 26$

X_4 = ABCADBCDABBCACDD. $Cmax = 25$

Paso 3: Selección.

Probabilidad de selección:

P_{x1}= 0'252 → 25,2% P_{x2}= 0'252 → 25,2%

P_{x3}= 0'244 → 24,4% P_{x4}= 0'252 → 25,2%

Pasamos ahora a dar cuatro números al azar entre 0 y 1 y elegimos las soluciones mediante el método de la ruleta:

0'36 (X_2) 0'81 (X_4) 0'49 (X_2) 0'28 (X_2)

Paso 4: Reproducción (cruce y mutación).

Pareja X_2 – X_4

-¿Cruzan? Número al azar: 35 → Sí cruzan.

>Cruce:

X_2: ABCADABCDBCBACDD → ABCAABCDBDCBACDD

X_4: ABCADBCDABBCACDD → ABCAABCDBDBCACDD

-¿Mutan? Número al azar: 99 → Sí mutan.

ABCAABCDBDCBACDD → ABCBABCDBDCAACDD (H_1)

ABCAABCDBDBCACDD → ABCACBCDBDBCAADD (H_2)

Pareja X_2 – X_2

-¿Cruzan? Número al azar: 77 → Sí cruzan.

X_2: ABCADABCDBCBACDD → AABCADBCDBCBACDD (H_3)

X_2: ABCADABCDBCBACDD → AABCADBCDBCBACDD (H_4)

-¿Mutan? Número al azar: 21 → No mutan.

Paso 5: Evaluación hijos.

Evaluamos los nuevos individuos (los hijos)

1º) H_1
ABCBABCDBDCAACDD → $Cmax = 25$

2º) H_2
ABCACBCDBDBCAADD → $Cmax = 25$

3º) H_3
AABCADBCDBCBACDD → $Cmax = 31$

4º) H_4
AABCADBCDBCBACDD → $Cmax = 31$

Paso 6: Sustitución elitista.

Pasamos todos los hijos menos el peor a la siguiente población.

Hijos: H_1 $Cmax = 25$

H_2 $Cmax = 25$

H_3 $Cmax = 31$

H_4 $Cmax = 31$ → Éste individuo se queda fuera.

Mejor individuo de la población inicial: X_1 con $Cmax = 25$ → éste como mejor individuo de la población inicial, sustituye a H_4.

Paso 7: ¿Condición de parada?

Como se ha llegado a la 4ª iteración, paramos el algoritmo y nos quedamos con la mejor solución encontrada hasta el momento:

X_1 = ABCBABCDBDCAACDD. $Cmax = 25$

X_2 = ABCACBCDBDBCAADD. $Cmax = 25$

X_3 = AABCADBCDBCBACDD. $Cmax = 31$

X_4 = ABCDAABCDBBCACDD. $Cmax = 25$

Capítulo 3

Algoritmo de Abejas.

3.1. Introducción.

El algoritmo de Colonia de Abejas Artificial (ABC[5]) o algoritmo de enjambre de abejas, fue introducido por primera vez en 2005 por Karaboga [44] enfocado a la resolución de problemas sin restricciones, y está inspirado en el comportamiento de las abejas en su búsqueda por alimento. Además esta técnica está especialmente diseñada para trabajar con varias variables y funciones continuas multimodales. Este algoritmo se encuadra dentro de la categoría de inteligencia de enjambre.

"La Inteligencia de enjambre es una rama de la Inteligencia artificial que estudia el comportamiento colectivo de los sistemas descentralizados, auto-organizados, naturales o artificiales" [71]. Se presentó en 1989 por G. Beni y J. Wang. Este tipo de algoritmos toman como inspiración la naturaleza, especialmente sistemas formados por una población de agentes que interactúan entre ellos y con el medio. El objetivo de los algoritmos de inteligencia colectiva es modelar los comportamientos de los mencionados agentes y sus interacciones con el entorno, para así obtener un comportamiento más complejo que pueda utilizarse para resolver problemas [4].

Dicha población no tiene un claro "líder" que guíe sus movimientos, sino que tiene un comportamiento de auto-organización, en el que los individuos

[5] De sus siglas en Inglés: *ABC- Artificial Bee Colony*.

desconocen el estado del colectivo, sin embargo gracias a la transferencia continua de información entre ellos logran una estrategia de grupo que los ayuda en la toma de decisiones.

En esta categoría, además del ABC, se encuentran entre otros, los conocidos algoritmo de colonia de hormigas (ACO), algoritmo de enjambre de partículas, algoritmo murciélago etc.

Volviendo al algoritmo de colonia de abejas, éste consta de tres elementos esenciales: posiciones de las fuentes de alimento, cantidades de néctar que tienen éstas últimas y los diferentes tipos de abejas. En el apartado siguiente explicaremos en qué consiste cada uno de ellos.

Aunque inicialmente el ABC se planteó para el estudio de problemas sin restricciones, a lo largo del tiempo se han introducido mejoras y variantes del algoritmo que pueden resolver solventemente problemas con restricciones y problemas de combinatoria [4].

El algoritmo de enjambre de abejas incorpora simultáneamente mecanismos de exploración y explotación, lo que lo hace idóneo para la resolución de problemas complejos, como el caso de estudio de este trabajo, el *job shop*. Sin embargo debido a su naturaleza continua, no se puede aplicar directamente a este tipo de problemas (de naturaleza discreta), esto se soluciona mediante un proceso de decodificación que transforma el vector de números reales con el que trabaja el algoritmo, en una permutación de enteros que representa una solución válida del problema. Esto se detallará más adelante [76].

3.2. Modelo Biológico.

El algoritmo define una colonia artificial formada por abejas, cuya misión es encontrar fuentes de alimento. El proceso de búsqueda de néctar es un proceso de optimización y el comportamiento que las abejas muestran se modeló como una heurística que consta de los siguientes elementos [4]:

Fuentes de alimento: puntos de extracción de alimento por parte de las abejas. Las fuentes de alimento están definidas por diferentes factores como su posición, su cantidad de néctar, su facilidad de extracción etc. Una fuente de alimento en el algoritmo, representa una solución válida del problema, y los factores que la definen determinan su calidad, relacionada con su valor para la función objetivo que se quiere optimizar.

Las abejas, representan al operador de variación, ya que cuando visitan una solución (fuente de alimento) buscan nuevas soluciones en su entorno. Las abejas se pueden clasificar en tres tipos según su función:

- Abejas recolectoras**:** son las encargadas de extraer el néctar de una fuente de alimento además de compartir toda la información acerca de la

fuente (ubicación, cantidad de néctar…) con la abeja observadora. El número de abejas recolectoras es igual a la cantidad de fuentes de alimento (una por cada fuente o lo que es lo mismo, un operador de cambio en cada solución). Aportan la capacidad de explotación.

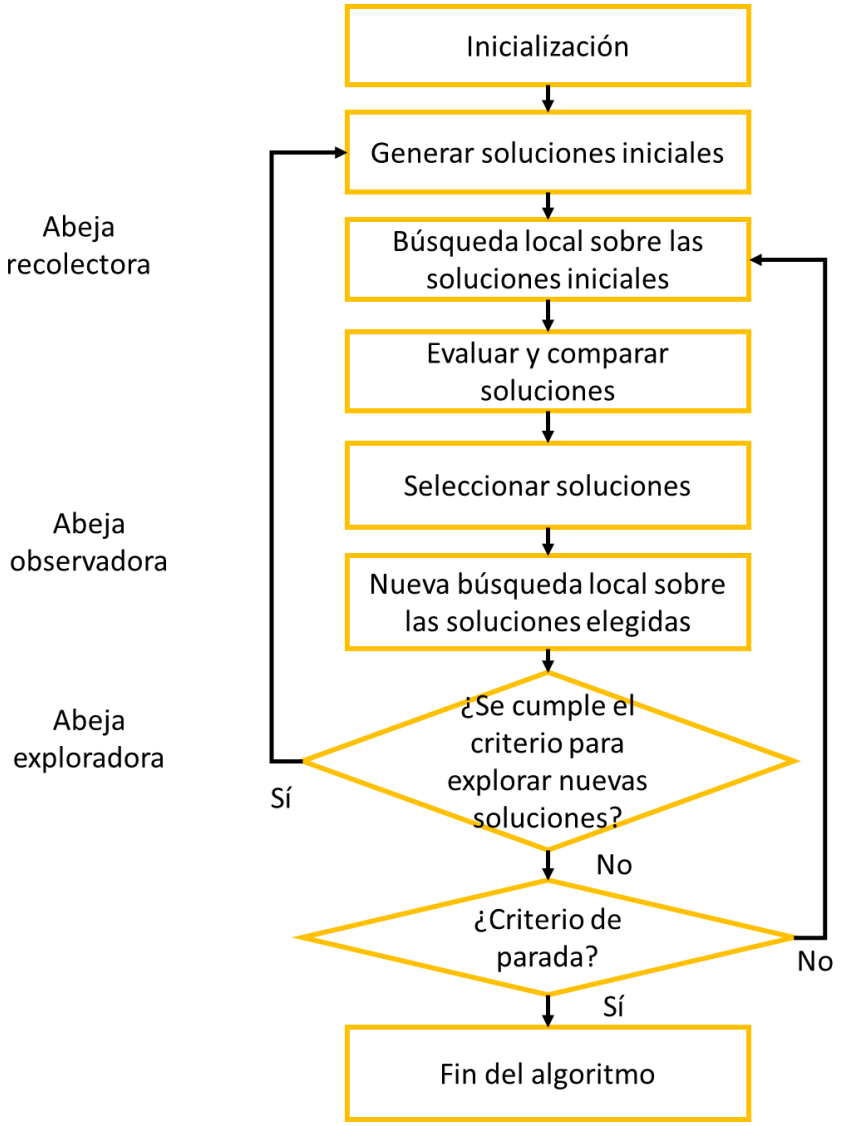
- Abejas observadoras: se sitúan en la colmena mientras presencian los movimientos de las abejas recolectoras esperando a que les pasen la información pertinente. Con esos datos, eligen una fuente de alimento en base a su calidad. Las abejas observadoras son las encargadas de dirigir la búsqueda hacia las mejores soluciones.

- Abejas exploradoras: cuando el néctar de una fuente de alimento se agota, la abeja recolectora encargada de su explotación, se convierte en una abeja exploradora en busca de una nueva fuente. Cuando encuentra una nueva fuente, la abeja retoma su trabajo de recolectora. Éste movimiento equivale a abandonar la búsqueda alrededor de una solución que no está aportando buenos resultados y dirigirla hacia una zona sin explorar del espacio de soluciones. La abeja exploradora aporta la capacidad de exploración.

Las abejas comparten la información por medio de una "danza", cuya duración indica la concentración de néctar (cuanto mejor sea una fuente más larga es la danza, ya que así tiene más posibilidades de ser vista por una abeja observadora), el ángulo respecto al sol señala su dirección y el número de movimientos en zig-zag la distancia desde la colmena [26].

3.3. Estructura

La estructura que a continuación se plantea, toma como base la obra de R. Zhang, S. Song y C. Wu [76] en la que los autores proponen el algoritmo de colonia de abejas artificial para minimizar el *retraso total ponderado* en un problema tipo *job shop*. Aunque en el presente trabajo, se ha modificado para minimizar el *makespan,* que se puede considerar como un caso particular de la formulación más general planteada en el mencionado artículo.



3.3.1. Inicialización.

Lo primero es definir los parámetros que necesita el algoritmo, en este caso hay que decidir cuál va a ser el número de fuentes de alimento SN, o lo que es lo mismo, el número de soluciones. Otro parámetro que usa, es D, el número de operaciones totales del problema. Este dato marca la dimensión del vector que veremos a continuación.

Una vez que ya están definidos los parámetros, el primer paso es inicializar el algoritmo mediante la creación de la población inicial. Se generan SN (número de fuentes de alimento) soluciones iniciales. Cada una de las soluciones se representa con un vector de dimensión D (número de operaciones total). Cada elemento del vector se obtiene con la siguiente expresión:

$$X_{i,d} = X_d^{min} + r(X_d^{max} - X_d^{min}) \qquad d = 1 \dots D \qquad i = 1 \dots SN$$

Siendo $X_{i,d}$ el elemento *d* de la solución *i;* X_d^{min} y X_d^{max} los límites inferior y superior respectivamente para la dimensión *d*; y *r,* un número aleatorio uniforme entre 0 y 1.

Por tanto esta expresión genera para cada elemento *d*, un número aleatorio entre X_d^{min} y X_d^{max}. Estos límites están definidos para la posición *d* del vector, y son los mismos para cualquier solución *i*. Esto quiere decir que una componente determinada siempre va a estar en el mismo rango dado por X_d^{max} y X_d^{min} , en cualquiera de las soluciones generadas; de forma que $X_{id} \in (X_d^{max}, X_d^{min})$.

Así nos queda el conjunto de la población inicial $X = \{\boldsymbol{X_i} \dots \boldsymbol{X_{SN}}\}$, siendo cada una de las soluciones $\boldsymbol{X_i} = (X_{i,1} \dots X_{i,D})$ un vector de D componentes.

3.3.2. Abeja recolectora: comienzo búsqueda local.

La siguiente etapa es "enviar" a cada fuente de alimento (solución) una abeja recolectora. La abeja recolectora buscará una nueva fuente de alimento en el entorno de la actual, lo que traducido al algoritmo significa que se realiza una pequeña modificación en la solución (búsqueda local). La nueva solución $\boldsymbol{V_i}$, se obtiene a partir de $\boldsymbol{X_i}$, de la siguiente manera:

$$V_{i,d} = X_{i,d} + r'(X_{i,d} - X_{k,d})$$

donde *d* es una componente del vector (puede valer desde 1 hasta D) obtenido aleatoriamente; X_k es otra solución perteneciente a la población inicial (con $K \neq i$) obtenida también de forma aleatoria; y *r'* es un número aleatorio uniforme comprendido entre -1 y 1.

3.3.3. Evaluar y comparar soluciones.

En este punto tenemos un conjunto de soluciones iniciales $X = \{\boldsymbol{X_1} \dots \boldsymbol{X_D}\}$ y un conjunto de nuevas soluciones $V = \{\boldsymbol{V_1} \dots \boldsymbol{V_D}\}$ obtenidas por búsqueda local, del entorno de las soluciones iniciales. La abeja recolectora trabajará en la mejor de las fuentes de alimento (soluciones) encontradas, por tanto es necesario primero evaluar cada una de ellas para más tarde poder realizar una comparativa.

Cada solución está formada por un vector de dimensión D, en el que cada elemento simboliza una operación del programa. Para obtener el programa que representa es necesario decodificar el vector, este proceso de codificación y decodificación se verá en apartados posteriores.

Una vez obtenido el programa asociado a una solución $\boldsymbol{X_i}$, se calcula su *makespan,* y se compara con el de su solución vecina correspondiente $\boldsymbol{V_i}$. Si $\boldsymbol{V_i}$ resulta de mayor calidad que $\boldsymbol{X_i}$, pasará a sustituirla y X_i, se olvidará.

3.3.4. Abeja observadora: selección de soluciones.

Cuando las abejas recolectoras terminan su búsqueda local, comparten la información obtenida con las abejas observadoras. Las abejas observadoras son las encargadas de elegir las fuentes de alimento sobre las que se realizará una nueva búsqueda local. Como se puede intuir, éstas tenderán a seleccionar las fuentes de alimento con más néctar, las mejores. O lo que es lo mismo, una solución tendrá más probabilidad de ser elegida cuanto mayor sea el valor de su función de aptitud[6], dicha probabilidad se obtiene a partir de la siguiente expresión:

$$p_i = \frac{f_i}{\sum_{j=1}^{SN} f_j}$$

siendo p_i la probabilidad de la solución *i* de ser elegida, calculada como el cociente entre f_i que es el valor de la función de aptitud de la solución *i*, y la suma de los valores de la función de aptitud, para las *SN* soluciones que el algoritmo utiliza.

Hay que tener en cuenta que utiliza el valor de la función de aptitud, y no el de la función objetivo. La función de aptitud está relacionada con la función objetivo y en nuestro caso viene dada por:

$$f_i = \frac{1}{C_i}$$

siendo C_i el valor de la función objetivo (*makespan*) para la función *i.*

La expresión usada pasa calcular la probabilidad, es la vista en el apartado de selección de algoritmos genéticos. Por lo que aquí también se podría usar (entre otros) el método de la ruleta para elegir una solución en base a su calidad.

El número de soluciones seleccionadas para la nueva búsqueda local es un parámetro que hay que decidir al comienzo del algoritmo. Algunos autores, definen este parámetro como la mitad de la población de soluciones.

3.3.5. Abeja Observadora: búsqueda local.

Una vez elegidas las fuentes de alimento, la abeja observadora conduce una nueva búsqueda local en el entorno de dichas soluciones. La forma tradicional de practicar la búsqueda es aplicando de nuevo la fórmula del apartado 3.3.2. de abejas recolectoras, aunque R. Zhang et al. [76] introducen una mejora en este paso mediante la aplicación de una búsqueda local más exhaustiva, utilizando un árbol de soluciones que se va ramificando según avanza la exploración.

[6] Función de aptitud: *fitness function* (en inglés).

En cada iteración de la búsqueda local se intercambian dos operaciones al azar correspondiente a los bloques del camino crítico de ese programa. Recordamos que se considera bloque a una secuencia de operaciones del camino crítico si se compone de al menos dos operaciones y se procesan en la misma máquina. A la herramienta de intercambio se le denomina operador *Swap.*

Para evitar que la búsqueda repita las mismas soluciones se crea una lista de intercambios prohibidos en la que se incluye las operaciones que ya se han intercambiado.

Algoritmo de búsqueda de árbol

Cada nodo del árbol representa una solución (un programa completo) y tiene asociado su valor en la función objetivo. Antes de comenzar este algoritmo de búsqueda, es necesario establecer unos parámetros:

L: número de niveles del árbol, el índice l indica el nivel actual, tal que *l* = 1…L.

η_1: número de soluciones que se ramifican en cada nivel.

η_2: número de ramificaciones que se realizan en cada solución.

Paso 1: Se establece el primer nivel del árbol (*l*=1), formado por la solución inicial (seleccionada en el paso anterior) a la que llamamos σ.

Paso 2: Se aplica el operador *swap*, que intercambia dos operaciones aleatorias pertenecientes a un bloque de σ, en η_1 localizaciones diferentes. Por tanto, se obtienen η_1 soluciones nuevas.

Paso 3: A las η_1 nuevas soluciones, se les aplica el operador *Swap* η_2 veces, obteniendo η_1x η_2 soluciones nuevas.

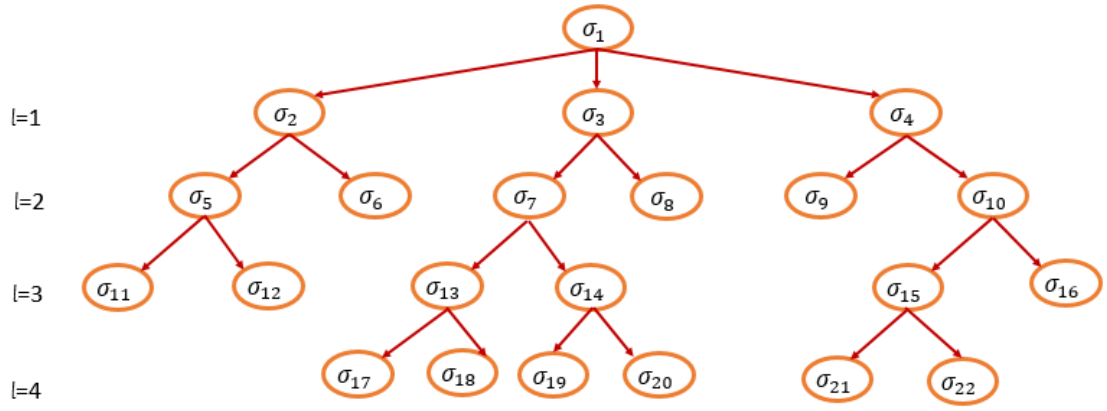
Paso 4: de esas η_1x η_2 soluciones, nos quedamos sólo con las $\eta_1$mejores (el número de soluciones que desechamos es η_1x $(\eta_2 - 1)$).

Paso 5: si *l*<L, pasamos al siguiente nivel y repetimos otra vez desde el paso 3.

Durante todo el algoritmo se guarda siempre la mejor solución encontrada hasta el momento (σ^*), y cuando se alcancen los L niveles, el resultado de la búsqueda, será σ^*.

Una vez finalizada la búsqueda, se compara la solución inicial σ con la mejor encontrada por el algoritmo de búsqueda de árbol σ^*, y si σ^* es mejor que σ, la nueva solución sustituye a la de partida.

Veamos un ejemplo en la siguiente figura de cómo quedaría el árbol con L=4, $\eta_1 = 3$ y η_2=2.



3.3.6. Abeja exploradora.

Si la calidad de una solución no se mejora en un número determinado de iteraciones, dicha fuente de alimento se abandona, y la abeja recolectora encargada de su explotación pasa a ser una abeja exploradora en busca de una nueva fuente de alimento. La nueva fuente de alimento (solución) se genera aleatoriamente igual que en el paso de inicialización.

Para comprobar si una solución candidata ha alcanzado el límite predeterminado de iteraciones, se asigna un contador a cada fuente de alimento. Dicho contador se incrementa si en una iteración no se mejora la solución actual [24].

Si no se ha llegado al número máximo de iteraciones, se vuelve a repetir todo el proceso desde el paso de la abeja recolectora, explorando así zonas no visitadas del espacio de soluciones. En todo momento el algoritmo almacena la mejor solución encontrada hasta el momento, que será la salida del proceso.

3.4. Codificación y decodificación de soluciones.

El algoritmo de colonia de abejas no trabaja directamente con un programa de operaciones, sino con una solución codificada que lo representa. Cada solución se define como un vector de números reales, con dimensión *D*, siendo *D* el número total de operaciones del problema:

$$\boldsymbol{X_i} = \{X_{i,1} \dots X_{i,D}\} \qquad d = 1 \dots D$$

A cada operación *d*, se le asigna el elemento del vector situado en la posición *d*. Por ejemplo, la operación 5, tendrá asignado el valor que aparezca en la posición 5 del vector.

Cada programa decodificado, lo representamos mediante una matriz *m* x *n,* siendo *m* el número de máquinas y *n* el número de trabajos (suponiendo que cada trabajo tenga tantas operaciones como máquinas haya y pase una

vez por cada una de ellas), de tal forma que una fila *k*, indique la secuencia de operaciones para esa máquina *k*.

Para llegar a esa representación, se aplica el algoritmo de decodificación:

Paso 1: ordenar de menor a mayor los valores del vector. Hay que tener en cuenta que cada valor está asociado a una operación, por lo tanto al ordenarlos, se obtiene una secuencia de operaciones $\boldsymbol{\pi_i}$.

Paso 2: Establecer $Q = \{1 \dots D\}$ como el conjunto de todas la operaciones y a partir de él, obtenemos el conjunto de todas las operaciones disponibles para ser ejecutadas en ese momento, el conjunto R. En la primera iteración, *R* contendrá a la primera operación de cada trabajo.

Paso 3: encontrar la operación i^* dentro del conjunto R que pueda terminar lo más pronto posible.

Paso 4: Establecer *B* como el conjunto de operaciones pertenecientes a *R*, que se ejecutan en la misma máquina que la operación i^* (ésta incluida también en *B*).

Paso 5: Eliminar del conjunto B las operaciones que no puedan empezar antes de lo que acabaría i^*.

Paso 6: Encontrar la operación perteneciente a B que aparezca antes en la secuencia $\boldsymbol{\pi_i}$, y programar dicha operación.

Paso 7: actualizar el conjunto *R* de operaciones disponibles (eliminando la que se acaba de programar y añadiendo su sucesora inmediata en la secuencia de trabajos).

Paso 8: si el conjunto *R* está vacío, el algoritmo de decodificación ha terminado, si todavía tiene alguna operación, se repite de nuevo desde el paso 3.

Para facilitar la aplicación de este algoritmo, se puede usar la siguiente tabla:

Máquina 1						Máquina m				
Op.	r_i	p_i	b_i	c_i		Op.	r_i	p_i	b_i	c_i
					...					

En la que:

Op_i: operación *i.*

r_i : fecha desde que la operación *i* está disponible, es decir, fecha en la que termina la operación inmediatamente predecesora del trabajo al que pertenece.

p_i: duración de la operación i.

b_i: fecha de comienzo de la operación *i.* Se establece en el momento en el que se programa la operación.

c_i: fecha de finalización de la operación *i.* $c_i = b_i + p_i$.

Para trabajar con la tabla, primero se sitúan en ella las operaciones disponibles (se rellenan los campos de operación, r_i y p_i), y se van programando siguiendo el algoritmo. Programar una operación significa asignarla una fecha de comienzo y de fin, es decir b_i y c_i.

Hay que tener en cuenta que una operación entra en el conjunto *R* de operaciones disponibles en el momento que el que se programa su inmediata predecesora, pero si la máquina a la que pertenece está ocupada, no podrá empezar hasta que no termine la operación que se está procesando. Esto quiere decir, que r_i es la fecha desde la que está disponible la operación *i*, pero la fecha en la que puede comenzar será el mayor valor entre r_i y el c_i más alto para esa máquina.

En el ejemplo al final del capítulo se visualiza mejor todo el procedimiento.

Algoritmo de codificación.

El algoritmo de codificación permite convertir un programa dado en una sucesión de operaciones, mediante la reversión del algoritmo de decodificación que acabamos de ver.

Para ello, dado un determinado programa, se crea un vector $\boldsymbol{S_i}$ de dimensión D, que al finalizar el proceso de codificación, equivaldrá a la secuencia $\boldsymbol{\pi_i}$ vista anteriormente**.** Dicho vector, al comienzo del algoritmo de codificación estará vacío. Las operaciones se van añadiendo según su fecha más temprana en la que pueden finalizar. Entre las operaciones que todavía no se han añadido al vector se elige la que más pronto puede finalizar y se introduce. Así la primera operación del vector será la que pueda terminar más pronto y la última operación, la que termine más tarde.

Como ya se ha señalado, el resultado de este proceso es la secuencia $\boldsymbol{\pi_i}$ de operaciones. Hay que tener en cuenta que un mismo programa puede estar representado por varias secuencias, pues ocurre a menudo que varias operaciones terminan a la vez, en ese caso se elige cualquiera de ellas para

añadir al vector. Pero esto no supone un problema, pues al decodificar cualquiera de los diferentes vectores obtenidos, se obtiene el mismo programa en cuestión.

3.5. Ejemplo Resuelto.

Para entender mejor el funcionamiento del algoritmo de colonia de abejas y visualizar de forma más clara cómo trabaja, se ha propuesto su aplicación para minimizar el *makespan* del siguiente problema tipo *job shop*:

Trabajo	A				B				C				D			
Operación	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Tiempo ejecución (p_i)	3	5	9	6	1	9	2	5	1	1	5	2	4	2	6	1
Máquina (M_i)	1	2	4	3	2	1	3	4	4	1	3	2	3	2	1	4

El problema tiene dimensión 4x4 (4 trabajos y 4 máquinas), teniendo en cuenta que cada trabajo pasa una única vez por cada máquina) con un total de 16 operaciones.

A continuación se muestra la primera iteración del algoritmo para ver cómo funciona paso a paso.

1. Inicialización.

Se generan las SN=3 soluciones iniciales. Cada solución es un vector de dimensión D=16, en el que sus elementos se generan a partir de la siguiente expresión dando cada vez un valor diferente para r.

$$X_{i,d} = X_d^{min} + r(X_d^{max} - X_d^{min}) \qquad d = 1 \dots 16 \qquad\qquad i = 1 \dots 3$$

Se obtiene:

$X_1 = [0.1, 1, 2.1, 3.1, 4.5, 5, 2.2, 3, 1.6, 0.7, 0.8, 2.9, 4.9, 4.4, 2.8, 0.3]$

$X_2 = [5, 3.1, 1.6, 2.8, 3.2, 0.1, 3.3, 2.9, 0.6, 0.5, 0.2, 3, 1, 1.5, 1.8, 2]$

$X_3 = [3, 1.5, 0.5, 4, 2.5, 2.6, 1.6, 1.4, 2.1, 0.1, 0.2, 5, 4.5, 4.3, 3.3, 1.1]$

2. Abeja recolectora: comienzo de la búsqueda local.

Obtenemos una nueva solución $\boldsymbol{V_i}$ a partir de cada $\boldsymbol{X_i}$ mediante una pequeña modificación aleatoria en una de sus componentes elegida al azar, de la siguiente forma:

$$V_{i,d} = X_{i,d} + r'(X_{i,d} - X_{k,d})$$

De manera que:

$$V_{ij} = X_{ij} \qquad j = 1 \dots D, \quad j \neq d$$

Aplicándolo queda:

$V_{1,1} = X_{1,1} + (-0.4)(X_{1,1} - X_{3,1})$ = 0.1 -0.4*(0.1-3) = 1.26

$V_1 =$ [1.26, 1, 2.1, 3.1, 4.5, 5, 2.2, 3, 1.6, 0.7, 0.8, 2.9, 4.9, 4.4, 2.8, 0.3]

$V_{2,8} = X_{2,8} + 0.8(X_{2,8} - X_{1,8})$ = 2.9 + 0.8*(2.9-3) = 2.64

$V_2 =$ [5, 3.1, 1.6, 2.8, 3.2, 0.1, 3.3, 2.64, 0.6, 0.5, 0.2, 3, 1, 1.5, 1.8, 2]

$V_{3,16} = X_{3,16} + 0.5(X_{3,16} - X_{1,16})$ = 1.1 + 0.9*(1.1-0.3) = 1.82

$X_3 =$ [3, 1.5, 0.5, 4, 2.5, 2.6, 1.6, 1.4, 2.1, 0.1, 0.2, 5, 4.5, 4.3, 3.3, 1.82]

3. Evaluar y comparar soluciones.

Siguiendo con la búsqueda local, para poder evaluar las soluciones, primero es necesario transformar el vector que se acaba de obtener en un programa válido. Para ello, aplicamos el algoritmo de decodificación, detallando el proceso para la operación $\boldsymbol{X_1}$.

1º iteración

Paso 1: ordenar de menor a mayor los elementos del vector. Se obtiene la secuencia de operaciones $\boldsymbol{\pi_1}$.

Componente	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$X_{1,j}$	0,1	1	2,1	3,1	4,5	5	2,2	3	1,6	0,7	0,8	2,9	4,9	4,4	2,8	0,3
$\pi_{1,j}$	1	16	10	11	2	9	3	7	15	12	8	4	14	5	13	6

Paso 2: Establecer el conjunto *R* de operaciones disponibles. R = {1, 5, 9, 13}, y colocarlas en la tabla. Para esta primera iteración las operaciones disponibles son las primeras de cada trabajo.

Máquina 1					Máquina 2					Máquina 3					Máquina 4				
Op.	r_i	p_i	b_i	c_i	Op.	r_i	p_i	b_i	c_i	Op.	r_i	p_i	b_i	c_i	Op.	r_i	p_i	b_i	c_i
1	0	3			5	0	9			13	0	2			9	0	1		

Paso 3: Encontrar la operación i^* del conjunto *R* que puede terminar lo antes posible. Esta sería la operación 9, que puede empezar en 0 y dura 1, por lo que podría terminar en 1.

Paso 4: Establecer *B* como el conjunto de operaciones pertenecientes a *R*, que se ejecutan en la misma máquina que la operación i^* (máquina 4), incluida también i^*. Se obtiene que B = {9} ya que es la única operación disponible para la máquina 4.

Paso 5: Eliminar del conjunto *B* las operaciones que no puedan empezar antes de lo que acabaría i^*. Como B sólo contiene a i^*, no se elimina ninguna operación.

Paso 6: Encontrar la operación perteneciente a *B* que aparezca antes en la secuencia $\boldsymbol{\pi_i}$, y programar dicha operación. Como B sólo contiene a i^*, programamos i^*, que es la operación 9.

Paso 7: actualizar el conjunto *R* de operaciones disponibles, eliminando la que se acaba de programar y añadiendo su sucesora inmediata, que en este caso es la operación 10. R = {1, 5, 10, 13}.

Máquina 1					Máquina 2					Máquina 3					Máquina 4				
Op.	r_i	p_i	b_i	c_i	Op.	r_i	p_i	b_i	c_i	Op.	r_i	p_i	b_i	c_i	Op.	r_i	p_i	b_i	c_i
1	0	3			5	0	9			13	0	2			9	0	1	0	1
10	1	1																	

Paso 8: si el conjunto *R* está vacío, el algoritmo de decodificación ha terminado, si todavía tiene alguna operación, se repite de nuevo desde el paso 3. El conjunto R todavía tiene operación, por lo que se vuelve al paso 3 y se repite el algoritmo hasta que estén programadas todas las operaciones.

2º iteración

Máquina 1					Máquina 2					Máquina 3					Máquina 4				
Op.	r_i	p_i	b_i	c_i	Op.	r_i	p_i	b_i	c_i	Op.	r_i	p_i	b_i	c_i	Op.	r_i	p_i	b_i	c_i
1	0	3			5	0	9			13	0	2	0	2	9	0	1	0	1
10	1	1			14	2	2												

En esta iteración i^* corresponde a la operación 13 y el conjunto B está formado también por la operación 13 únicamente.

3º iteración

La operación que podría terminar antes es la operación 10, por tanto i^*=10, a la que corresponde la máquina 1. En esa máquina están las operaciones 1 y 10, por tanto B={1,10}, y no se elimina ninguna, puesto que la operación 1 puede empezar antes (en 0) de lo que terminaría la 10. Por tanto programamos la que antes aparezca en la secuencia, que en este caso es la operación 1:

$\boldsymbol{\pi_1}$ = {1 16 10 11 2 9 3 7 15 12 8 4 14 5 13 6}

Máquina 1					Máquina 2					Máquina 3					Máquina 4				
Op.	r_i	p_i	b_i	c_i	Op.	r_i	p_i	b_i	c_i	Op.	r_i	p_i	b_i	c_i	Op.	r_i	p_i	b_i	c_i
1	0	3	0	3	5	0	9			13	0	2	0	2	9	0	1	0	1
10	1	1			14	2	2												
					2	3	3												

Así se continúa hasta que todas las operaciones queden programadas o lo que es lo mismo, hasta que tengan asignada una fecha de inicio y de fin.

Al final del algoritmo, la tabla completa queda de la siguiente forma:

Máquina 1					Máquina 2					Máquina 3					Máquina 4				
Op.	r_i	p_i	b_i	c_i	Op.	r_i	p_i	b_i	c_i	Op.	r_i	p_i	b_i	c_i	Op.	r_i	p_i	b_i	c_i
1	0	3	0	3	5	0	9	13	22	13	0	2	0	2	9	0	1	0	1
10	1	1	3	4	14	2	2	9	11	11	4	5	4	9	3	9	6	9	15
15	11	6	11	17	2	3	6	3	9	4	15	6	15	21	16	17	1	17	18
6	22	9	22	31	12	11	2	11	13	7	31	2	31	33	8	33	5	33	38

Por tanto el *makespan* para este programa es 38, puesto que es la fecha en la que se ejecuta la última operación.

Evaluamos de la misma forma el resto de soluciones y obtenemos:

$Cmax_{X_1}$ = 38 $\qquad$ $Cmax_{V_1}$ = 38

$Cmax_{X_2}$ = 44 $\qquad$ $Cmax_{V_2}$ = 44

$Cmax_{X_3}$ = 28 $\qquad$ $Cmax_{V_3}$ = 28

Una vez evaluadas las comparamos entre sí (la solución original con su vecina) y nos quedamos con la mejor. En este caso, las soluciones nuevas no producen mejora, por lo que continuamos con las originales. Así, la población para el siguiente paso la forman las soluciones X_1, X_2 y X_3.

4. Abejas observadoras: selección de soluciones.

Primero hay que calcular la probabilidad P_i de cada solución X_i de ser elegida, a partir de la siguiente expresión:

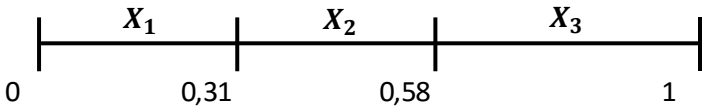
$$P_i = \frac{f_i}{\sum_{j=1}^{SN} f_j} \qquad \text{con} \quad f_i = \frac{1}{Cmax_i},$$

siendo $Cmax_i$ el valor del makespan de la solución i.

Esto nos da:

$$P_{X_1} = 0.31 \quad P_{X_2} = 0.27 \quad P_{X_3} = 0.42$$

Con representación lineal:

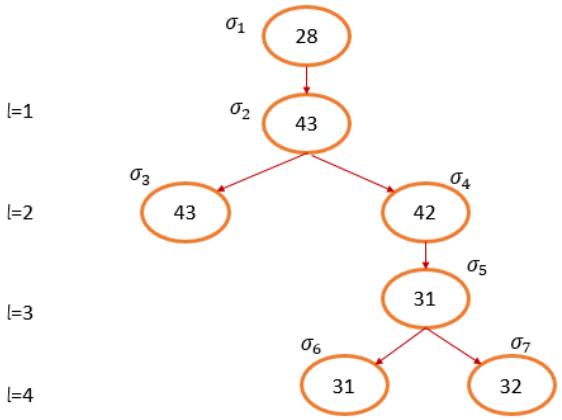


Como tenemos una población de 3 soluciones, se ha determinado en este ejemplo que sólo una de ellas pase a la nueva búsqueda local. Esto se realiza generando un número al azar y analizando en qué sección queda. El número al azar es 0.83, por tanto la solución elegida es X_3.

5. Abejas observadora: búsqueda local.

Para mejorar la eficacia de la búsqueda local se aplica el algoritmo de búsqueda de árbol. Para ello, se han elegido los siguientes parámetros:
L=4: número de niveles del árbol, tal que *l* = {1...L}.
$\eta_1 = 1$: número de soluciones que se ramifican en cada nivel.
$\eta_2 = 2$: número de ramificaciones que se realizan en cada solución.

Después de aplicar el algoritmo se obtiene el árbol de soluciones mostrado a continuación:



Como se puede observar, el árbol no se ha desarrollado del todo, porque por ejemplo en el nivel dos, la solución se debería ramificar en dos soluciones, y sólo lo hace en una. Esto se debe a que al ser un problema pequeño, muchas veces no se tienen suficientes bloques en los que aplicar el operador de intercambio de operaciones.

Capítulo 4

La Búsqueda Tabú.

4.1. Introducción.

El algoritmo de búsqueda tabú es un procedimiento metaheurístico primeramente introducido por Glover en 1987 [33], que permite encontrar soluciones cuasi-óptimas a un problema. La búsqueda tabú está dentro de la categoría de búsqueda local o búsqueda por entornos, pero introduce una nueva característica, el uso de estructuras de memoria, lo cual aumenta su rendimiento.

Paralelamente al trabajo de F. Glover, P. Hansen sugirió una heurística muy similar a la búsqueda tabú llamada *steepest ascent mildest descent* [38]. Mientras que Glover siguió desarrollando el método en publicaciones posteriores [35]. Entre las aplicaciones de la búsqueda tabú al problema concreto del job shop se encuentra el trabajo de E. Nowicki y C. Smutnicki [50].

A diferencia de los dos métodos vistos anteriormente, que son técnicas probabilísticas, la búsqueda tabú es un procedimiento determinista; esto quiere decir, que con las mismas entradas siempre se obtienen las mismas salidas, ya que el azar no interviene.

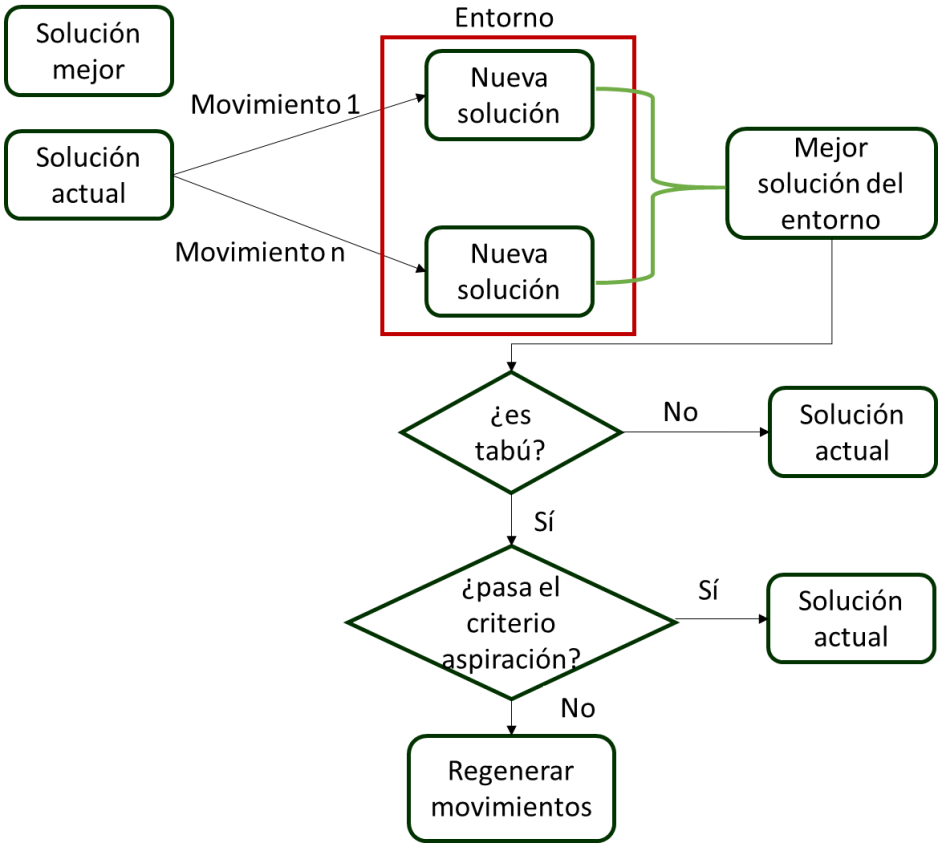
Un concepto básico de este método son los *movimientos*, que se pueden definir como una perturbación particular que transforma una solución en otra. El conjunto de movimientos posibles asociados a una solución define su *entorno* o *vecindad*. Partiendo de una solución, el algoritmo elige la mejor de su entorno, a la vez que clasifica como tabú la solución desde la que parte, es decir, la que acaba de visitar [50]. Esto evita quedar atrapado en óptimos locales y favorece la exploración de nuevas zonas del espacio de soluciones,

las cuales, aunque cabe la posibilidad de que sean peores que las zonas ya exploradas, pueden conducir hacia el óptimo global del problema.

La búsqueda tabú se basa en la premisa de que, una búsqueda inteligente debe incorporar *memoria adaptativa* y *exploración sensible*. La importancia de la exploración sensible en la búsqueda tabú, se deriva de la suposición de que una mala elección estratégica puede aportar más información que una buena elección al azar. Esto se debe, a que una decisión tomada estratégicamente aunque no dé buenos resultados, puede dar pistas sobre cómo mejorar dicha estrategia. La memoria adaptativa es útil para recordar las decisiones tomadas y aprovechar toda la información para guiar la búsqueda hacia soluciones óptimas [56]. La estructura de memoria más importante es la *lista tabú*, la cual se explicará más adelante.

En principio, las soluciones tabú no están permitidas, pero para hacer el método más flexible, se introduce el *criterio de aspiración,* con el cual, si una solución es tabú, pero satisface dicho criterio, puede ser elegida.

La imagen siguiente muestra un esquema del funcionamiento general de la búsqueda tabú. En apartados posteriores se verá una estructura más concreta y aplicada al problema del *job shop*.



Funcionamiento general de la búsqueda tabú [63].

4.2. Estructuras de memoria.

Como se ha comentado antes, las estructuras de memoria son un aspecto clave en la búsqueda tabú. Participan en la generación del entorno y ayudan a organizar la búsqueda a través del espacio de soluciones.

Mientras que otras técnicas sólo memorizan la mejor solución encontrada, la búsqueda tabú almacena la información pertinente sobre las soluciones visitadas.

Se puede distinguir entre memoria de corto plazo y memoria de largo plazo, cada una basada en una estrategia diferente.

* Memoria de corto plazo [56].

La memoria de corto plazo o también conocida como memoria basada en hechos recientes, es la más usada. Guarda la información sobre las soluciones visitadas recientemente y guía la búsqueda de forma inmediata desde la primera iteración del algoritmo.

La memoria utilizada puede ser *explícita* o *basada en atributos*. En la primera de ellas, se conservan las soluciones completas, mientras que en la basada en atributos sólo se conserva el movimiento que ha llevado a esa solución. Los atributos pertenecientes a las últimas soluciones por las que ha pasado el algoritmo, se denominan tabú-activos. La memoria basada en atributos además de evitar soluciones recientemente visitadas, también evita otras que tienen los mismos elementos tabú-activos que las últimas soluciones. Esto provoca que la búsqueda tenga un carácter muy agresivo; para flexibilizar el algoritmo, se introducen los llamados criterios de aspiración. Dichos criterios permiten realizar un movimiento aunque sea tabú, cuando el valor de la solución sea mejor que el mejor encontrado.

La estructura de memoria a corto plazo más importante es la lista tabú. Por ende la lista tabú es una estructura de memoria de corto plazo que contiene información sobre las últimas soluciones visitadas. Puede ser un tipo de memoria explícita o basada en atributos, lo cual quiere decir que la lista tabú puede guardar directamente las últimas soluciones visitadas o los últimos movimientos realizados. Para el caso concreto del problema del *job shop*, se utiliza más frecuentemente la memoria basada en atributos, y será la comentada en el presente trabajo.

El alcance de la regresión de la lista tabú queda definido por lo que se denomina *tenencia tabú*. La tenencia tabú, indica el tiempo que un atributo permanece en la lista tabú, o lo que es lo mismo, la longitud de la lista tabú. Comenzando con una lista vacía, se van añadiendo elementos en cada iteración de la búsqueda, hasta que alcanza su capacidad máxima (tenencia tabú), a partir de ese momento, cuando se añade un nuevo elemento a la lista, el elemento más antiguo sale y ya no es tabú.

* Memoria de largo plazo.

En muchas ocasiones, la estructura de memoria a corto plazo es suficiente para que el algoritmo dé buenos resultados, no obstante, el uso de memoria de largo plazo aumenta significativamente la potencia de la búsqueda tabú. Anteriormente se ha comentado que la memoria a corto plazo actúa inmediatamente sobre la búsqueda; en cambio, la memoria a largo plazo actúa a posteriori después de que se hayan realizado varias iteraciones [5].

Este tipo de memoria se utiliza para continuar con la búsqueda desde otra solución que no pertenece al entorno de la solución actual. Hay dos tipos de estrategias de memoria a largo plazo dependiendo del objetivo buscado:

-Estrategias de intensificación.

Se basan en una reinicialización de la búsqueda hacia regiones del espacio prometedoras, partiendo desde una solución ya visitada anteriormente y continuando la búsqueda en otra dirección diferente a la que se tomó en su momento.

La estrategia de intensificación se puede realizar cada un cierto número (fijo) de iteraciones o cuando se llega al final de un camino (cuando se da un ciclo, en el que la búsqueda recorre una y otra vez las mismas soluciones).

Algunas opciones de aplicación son:

-Reiniciar la búsqueda desde la mejor solución encontrada.

-Utilizar una lista de capacidad limitada en la que se guardan las iteraciones en las que se mejora la mejor solución encontrada en ese momento. Cuando se guarda una solución nueva, sale de la lista la más antigua. La búsqueda se reinicia desde el último elemento de la lista, es decir, el más reciente.

Cuando se aplica una estrategia de memoria a largo plazo, se puede proceder de dos formas diferentes respecto a la memoria a corto plazo. Una es vaciar su estructura de memoria y la otra opción es retomar la lista tabú que había en el momento en el que se pasó anteriormente por esa solución.

Un ejemplo de estrategia de intensificación es la estrategia del *back jump tracking,* de la que se hablará más adelante.

-Estrategias de diversificación.

Las estrategias de diversificación proponen continuar la búsqueda desde una solución no visitada anteriormente, para así poder explorar una nueva zona del espacio de soluciones.

Lo que se pretende es dirigir la búsqueda hacia zonas fuertemente contrastadas con las regiones ya exploradas. Esto se puede conseguir mediante diversos mecanismos:

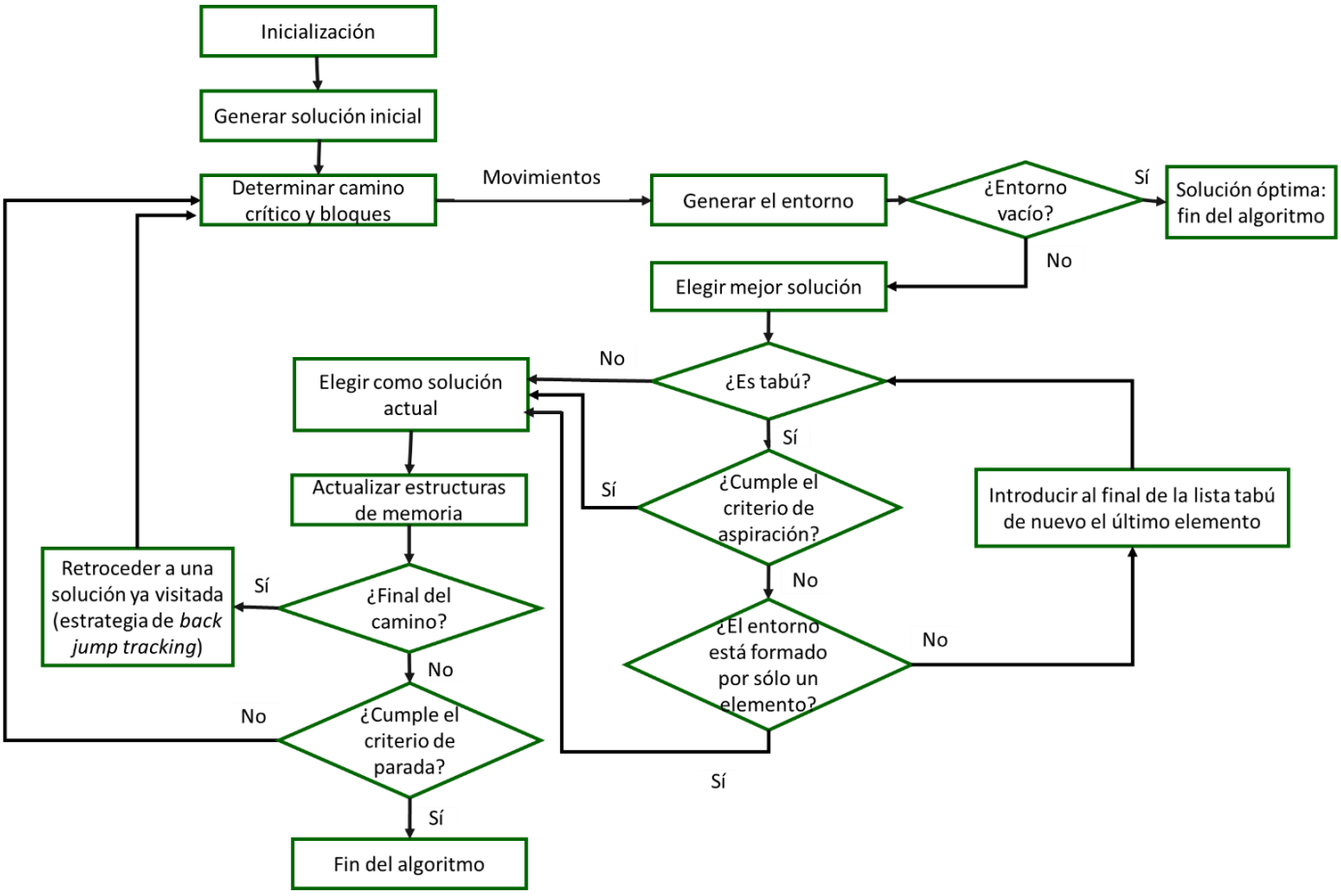
-Cambiar las reglas de selección endureciendo las restricciones en las condiciones tabú, restringiendo así un elevado número de soluciones.

-Penalizar las soluciones que usen atributos muy frecuentes en las zonas ya visitadas. De esta forma se favorece que la búsqueda se dirija hacia soluciones con otros atributos distintos.

Estos mecanismos sólo se aplican cuando se ejecuta una estrategia de diversificación, en el momento en el que la búsqueda se reinicia desde una nueva solución, se vuelven a relajar las condiciones.

4.3. Estructura: búsqueda tabú aplicada al job shop

A continuación se propone una estructura del algoritmo de búsqueda tabú aplicada para el problema del *job shop*, basada en el artículo de Nowicki y Smutnicki [50]:



4.3.1. Solución inicial.

Para inicializar el algoritmo, el primer paso es generar una solución inicial desde la cual comenzar la búsqueda. Existen varias formas de obtener una solución inicial, la primera de ellas es hacerlo aleatoriamente, otra es mediante un algoritmo heurístico sencillo que consiga una solución de calidad. Lo ideal es comenzar la búsqueda desde una buena solución, ya que de este modo se reduce el tiempo de procesamiento del algoritmo. No obstante, si se parte de una solución de baja calidad no supone un gran problema, ya que el algoritmo se encarga de mejorar el resultado en cada iteración.

Un ejemplo de heurística para obtener una solución inicial en el problema concreto del *job shop*, es el *algoritmo de inserción* (INSA, de sus siglas en inglés *insertion algorithm).* La idea en la que se basa, es programar primero las operaciones del trabajo cuyo tiempo de ejecución sea mayor, para luego programar el resto de operaciones intentando que no se incremente el tiempo total del proceso [45].

Una técnica de generación de la solución inicial aleatoriamente es la vista en el capítulo del algoritmo de colonia de abejas, en la que se genera un vector de números aleatorios con tantas componentes como operaciones tenga el problema y éstas se van colocando dependiendo del momento en el que están disponibles y de su orden en dicho vector.

4.3.2. Generación del entorno.

En el apartado 1.5.3., hablamos de la definición de *bloque*: una secuencia de operaciones del camino crítico es un bloque si contiene al menos dos operaciones y dichas operaciones se procesan en la misma máquina.

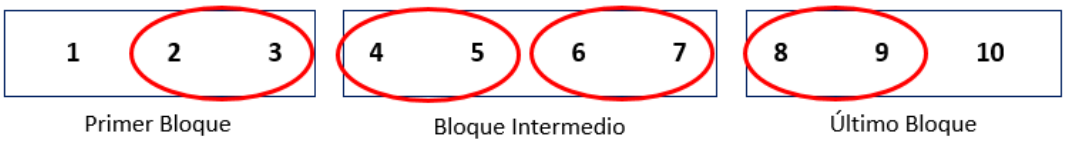
Por tanto, el primer paso es determinar el camino crítico de una solución, y a partir de él, sus bloques.

En la introducción de este capítulo, se definía un *movimiento* como una perturbación que transforma una solución en otra. En el caso del *job shop*, dicha perturbación viene dada por el intercambio de dos operaciones en la secuencia de máquinas del programa. Es decir, partiendo de un orden en la secuencia de máquinas determinado (un programa), se realiza un movimiento mediante el intercambio del orden de dos operaciones consecutivas en la misma máquina con el fin de obtener otro programa diferente.

Un aspecto importante es elegir los movimientos estratégicamente, pretendiendo llegar a buenas soluciones. Para ello, hay que tener en cuenta que el intercambio de dos operaciones que se realizan en la misma máquina y no pertenecen al camino crítico, no reduce el *makespan* [50]. Por este motivo todos los movimientos se realizarán en operaciones pertenecientes al camino crítico, es decir intercambiando las operaciones de los bloques.

Por tanto el entorno de una solución queda definido por todas las soluciones a las que se llega intercambiando operaciones consecutivas en los bloques de la solución de partida.

Dentro de los movientos posibles, hay unos más convenientes que otros. Sólo se intercambiarán las dos primeras operaciones de un bloque y las dos últimas. Además en el primer bloque sólo se hará el intercambio en las dos últimas operaciones y en el último bloque, solamente en las dos primeras. En los bloques intermedios, si su longitud lo permite (si hay tres o más operaciones), se hará tanto en las dos primeras operaciones del bloque como en las dos últimas. La imagen siguiente muestra un ejemplo de estas condiciones:



La imagen representa el camino crítico de un programa. Dicho camino crítico contiene 10 operaciones y 3 bloques. En un círculo rojo se marcan los movimientos que se efectuarían [50].

Hay que señalar que es debido al reducido tamaño del entorno por lo que la búsqueda tabú funciona tan rápido. Porque en vez de analizar todos los posibles intercambios entre operaciones o realizar los intercambios al azar, la búsqueda tabú sólo evalúa los intercambios de operaciones que tienen más probabilidades de mejorar el resultado. De esta forma se optimiza la capacidad computacional del algoritmo.

Es necesario señalar que Nowicki y Smutnicki [50] en su artículo demuestran que otros movimientos no mejoran el valor del *makespan*.

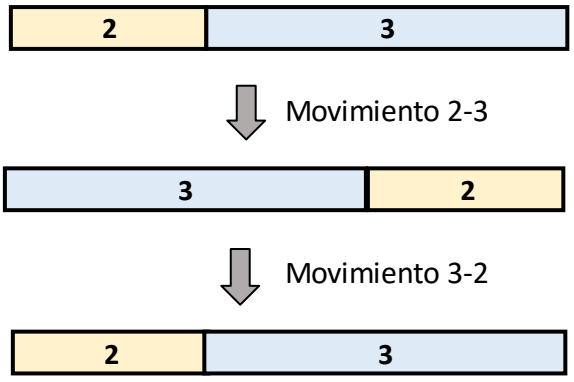
Una vez realizados los movimientos que dan lugar a los nuevos programas (soluciones), el entorno queda definido. Seguidamente, se calcula el camino crítico asociado y el valor del makespan de cada uno de estos programas.

4.3.3. Lista tabú

Cuando el entorno de la solución de partida ha sido generado, hay que determinar la siguiente solución por la que continuará la búsqueda. El criterio de selección es elegir la solución del entorno que optimice el valor de la función objetivo, que en este caso será la solución que minimice el makespan. Pero antes de tomarla como la solución definitiva desde la que comienza la siguiente iteración, primero hay que verificar ciertos aspectos.

El primero de ellos es comprobar que el movimiento que lleva a la mejor solución no está en la lista tabú. Recordamos que en la lista tabú se van

almacenando los últimos movimientos realizados. Hay que tener en cuenta que la forma de reflejar los movimientos efectuados es introducir el orden inverso a como estaban las operaciones en la solución inicial. Por ejemplo, en el caso del apartado anterior en el que se tenían 10 operaciones con tres bloques; si se realiza el movimiento entre las operaciones 2 y 3, la lista tabú contendría el movimiento 3-2.



Como se puede apreciar en la imagen, se parte del orden 2-3, y mediante la aplicación del movimiento 2-3, se llega al orden 3-2. Si lo que se quiere evitar es volver a la secuencia inicial, el movimiento tabú debe ser 3-2. Por tanto en la lista tabú se guarda el inverso al realizado.

Como se ha comentado, es necesario evaluar si el movimiento que conduce a la mejor solución del entorno se encuentra en la lista; si no es así, se elige directamente y se pasa a la siguiente iteración después de añadirlo en la última posición de la lista (los demás movimientos se desplazan de forma que el más antiguo se cae de la lista).

En cambio si el movimiento mejor del entorno sí se encuentra en la lista tabú, será necesario evaluar si cumple o no el criterio de aspiración, si satisface este criterio se elige para comenzar la siguiente iteración. En caso contrario se desarrolla una estrategia que se detalla en el apartado siguiente.

4.3.4. Criterio de aspiración.

Cuando el movimiento seleccionado se encuentra en la lista tabú no se descarta directamente, sólo en caso de que no supere el criterio de aspiración. Evidentemente, se tiene predilección por las buenas soluciones cuyos movimientos no son tabú-activos, pero esta condición es demasiado exigente y parte de la eficacia de la búsqueda tabú se basa en la relajación de dichas exigencias. Por tanto, el criterio de aspiración permite al algoritmo ser más flexible.

Una solución supera el criterio de aspiración si el valor de su función objetivo es mejor que el de cualquier solución vista anteriormente. Para nuestro

caso, una solución cumple el criterio de aspiración si el valor de su *makespan* es más pequeño que el de la mejor solución encontrada hasta el momento.

Teniendo en cuenta todo lo que se acaba de ver, se pueden clasificar los movimientos en tres clases [50]:

-*No prohibidos*: no pertenecen a la lista tabú.

-*Prohibidos pero ventajosos*: movimientos que están en la lista tabú, pero consiguen mejorar el mejor valor encontrado para el *makespan*.

-*Prohibidos y no ventajosos*: movimientos de la lista tabú que además no mejoran el mejor valor del *makespan*.

Por tanto, nos podemos encontrar ante tres situaciones diferentes:

-Si existen movimientos *no prohibidos* y movimientos *prohibidos pero ventajosos*, se elige al que conduzca a la mejor solución; indistintamente del tipo de movimiento.

-Si no se tiene ningún movimiento *no prohibido* ni ninguno *prohibido pero ventajoso*, y existe al menos uno *prohibido y no ventajoso*, se plantean dos opciones. Si el entorno está formado por un único elemento, se elige esa solución. En cambio, si el entorno tiene varios movimientos, el último elemento de la lista tabú se añade de nuevo a la lista reiteradamente hasta que aparezca un movimiento *no prohibido* que pueda ser elegido.

-Si no se tienen movimientos de ningún tipo, es indicativo de que se ha alcanzado la solución óptima. Esto se verá en el apartado de criterios de parada.

Al igual que antes, una vez elegido el movimiento pertinente, se añade a la lista tabú y se continúa la búsqueda en la siguiente iteración a partir de la nueva solución elegida.

4.3.5. Estrategia de Back Jump Tracking.

La estrategia de *back jump tracking* o *seguimiento de salto hacia atrás* en español, es una estrategia de diversificación (visto en el apartado de estructuras de memoria de largo plazo) y se aplica en el momento en que un camino de búsqueda termina. Se considera que un camino ha llegado a su fin en el momento en el que se detecta un ciclo, es decir, iterando, se pasa una y otra vez por las mismas soluciones. La estrategia del *back jump tracking* permite salir del ciclo. La forma de hacerlo es retrocediendo un tramo del camino y en ese punto ramificar el camino emprendiendo rumbo en otra dirección diferente a la tomada previamente.

Para ello es necesario crear una lista de longitud prefijada en la que se guardan las soluciones que mejoran el mínimo valor del *makespan* encontrado hasta ese momento. En dicha lista no sólo se guardan las operaciones, sino toda la información relacionada con ellas. La lista contiene:

-La solución que mejora el mejor *makespan* conocido en ese momento.

-La lista tabú de esa iteración.

-El entorno de la solución, pero excluyendo el movimiento que se eligió en ese momento.

Cuando se llega al final de un camino, se examina la lista del *back jump tracking*, y se escoge el último elemento de la lista para volver sobre él. Puesto que se ha eliminado el movimiento que elegiría, se obliga al algoritmo a tomar una nueva dirección, generando de esta forma una ramificación en el camino [50].

4.3.6. Criterio de parada.

Pueden existir diversos criterios de parada. El más utilizado es parar cuando se alcanza un número dado de iteraciones. Otra opción es fijar un número límite de iteraciones desde la última mejora del mejor valor encontrado. En ambos casos, cuando se cumple el criterio de parada el algoritmo se detiene y devuelve la mejor solución encontrada, que no tiene por qué ser la óptima.

Otro criterio es detener el algoritmo cuando se ha alcanzado la solución óptima, por lo que obviamente, no es necesario continuar la búsqueda. Pero, ¿cómo sabemos si una solución es la óptima? Un indicativo de ello es cuando no existe ningún tipo de movimiento disponible para realizar. Esto quiere decir que el camino crítico está compuesto únicamente por operaciones pertenecientes al mismo trabajo (ya que si no fuera así, existiría algún bloque y por lo tanto habría movimientos disponibles). Si el camino crítico está formado únicamente por las operaciones de un mismo trabajo, implica que no se puede reducir más la duración del *makespan*.

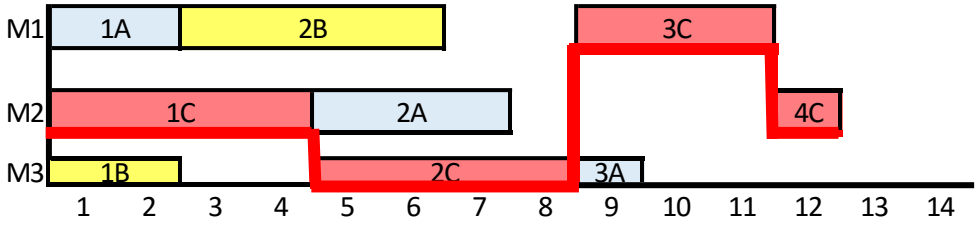
Hay que señalar que el hecho de que no existan movimientos implica que se ha alcanzado el óptimo, pero no al revés. Es decir, un programa puede ser el óptimo aun teniendo movimientos disponibles.

Las condiciones de parada que se acaban de citar son compatibles entre sí y pueden usarse conjuntamente en un mismo algoritmo.

A continuación se muestran dos ejemplos que ilustran lo que se acaba de comentar acerca de la solución óptima:

TRABAJO	OPERACIÓN	TIEMPO EJECUCIÓN	MÁQUINA
A	1A	2	M1
	2A	3	M2
	3A	1	M3
B	1B	2	M3
	2B	4	M1
C	1C	4	M2
	2C	2	M3
	3C	3	M1
	4C	1	M2

La solución óptima para este problema es:

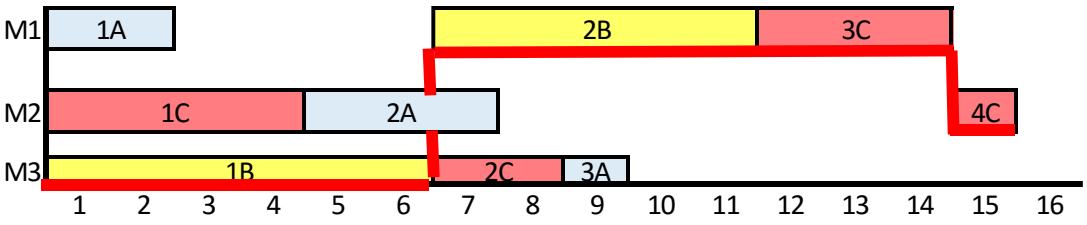


La duración total de este programa no se puede minimizar más ya que únicamente depende de la duración de las actividades del trabajo C.

Sin embargo, veamos otro ejemplo muy similar al anterior, en el que sólo cambia la duración de las actividades del trabajo B:

TRABAJO	OPERACIÓN	TIEMPO EJECUCIÓN	MÁQUINA
A	1A	2	M1
	2A	3	M2
	3A	1	M3
B	1B	6	M3
	2B	5	M1
C	1C	4	M2
	2C	2	M3
	3C	3	M1
	4C	1	M2

La solución óptima para este problema es la siguiente:



Esta solución es la óptima y aun así, su camino crítico tiene bloques y existen movimientos disponibles para hacer. Lo cual corrobora el hecho de que el que no haya bloques supone que una solución es óptima, pero no al revés.

4.4. Ejemplo resuelto.

Para entender mejor el funcionamiento del algoritmo de búsqueda tabú y visualizar de forma más clara su funcionamiento, se ha propuesto su aplicación para minimizar el *makespan* del siguiente problema tipo *job shop*:

Trabajo	A				B				C				D			
Operación	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Tiempo ejecución (p_i)	3	5	9	6	1	9	2	5	1	1	5	2	4	2	6	1
Máquina (M_i)	1	2	4	3	2	1	3	4	4	1	3	2	3	2	1	4

El problema tiene dimensión 4x4 (4 trabajos y 4 máquinas), teniendo en cuenta que cada trabajo pasa una única vez por cada máquina) con un total de 16 operaciones.

A continuación se muestran las dos primeras iteraciones del algoritmo para ver cómo funciona paso a paso.

1º iteración

1. Inicialización.

Antes de comenzar la búsqueda, es necesario definir el parámetro que indica el número máximo de iteraciones. En este ejemplo se tomará el valor de 50 iteraciones. Otro parámetro necesario es el número máximo de iteraciones sin mejora, que tomaremos como 10.

También es necesario inicializar las estructuras de memoria. Como es lógico en este punto ambas listas estarán vacías:

Lista tabú = { Ø }

Longitud lista tabú = 6

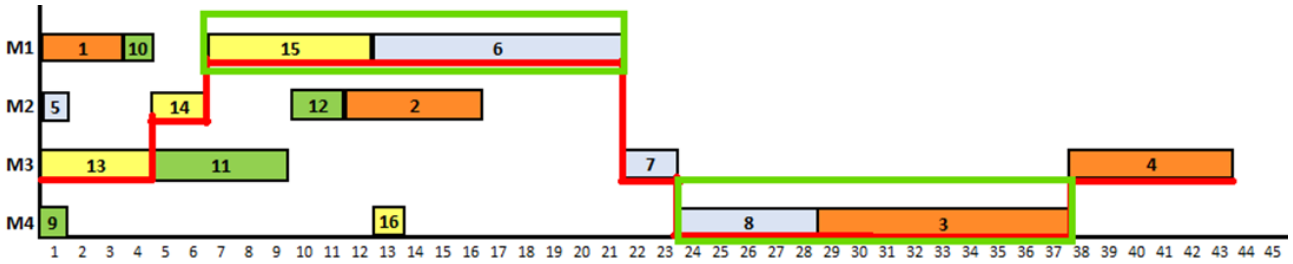
Lista *back jump tracking* = { Ø }

Longitud lista back jump tracking = 4

Y la solución mejor encontrada en este momento se toma como valor infinito.

2. Solución inicial.

La solución inicial se ha obtenido de forma aleatoria mediante el método descrito en el algoritmo de colonia de abejas, consiguiendo como resultado el siguiente programa, cuyo *makespan* es de 43.



3. Generar entorno.

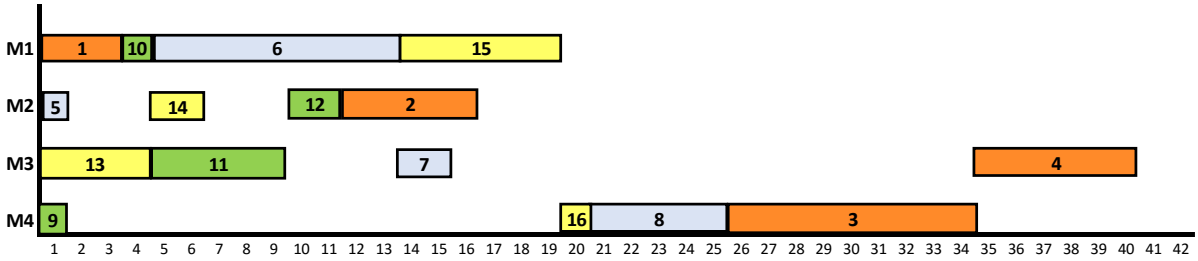
El camino crítico del programa está marcado con una línea roja en el diagrama de Gantt y está compuesto de las siguientes operaciones: 13-14-15-6-7-8-3-4.

Los bloques están recuadrados en color verde y son los siguientes:

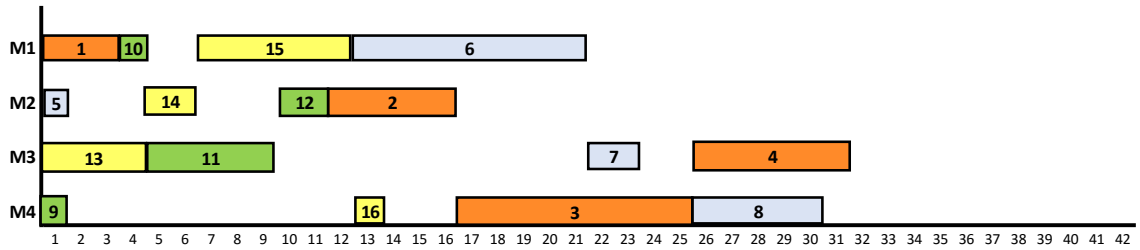
-Bloque 1: formado por la operaciones 15 y 6.
-Bloque 2: formado por la operaciones 8 y 3.

El entorno queda definido por los posibles movimientos, que en este caso, como los bloques sólo tienen dos operaciones cada uno, serán las que se intercambien. Los posibles movimientos con sus respectivos programas son:

Intercambio de las operaciones 15-6, se obtiene un programa cuyo *makespan* es 40:



Intercambio de las operaciones 8-3, se obtiene un programa cuyo *makespan* es 31:



La solución del entorno de mayor calidad viene dada por el movimiento 8-3.

Como el entorno no es vacío, no estamos en la solución óptima.

4. **Comprobar si es tabú**

Al ser la primera iteración, la lista tabú todavía no tiene ningún elemento, por lo que el movimiento 8-3 no está en la lista. Eso quiere decir que se puede elegir como solución inicial de la siguiente iteración.

5. **Criterio de aspiración**

Como no está en la lista tabú, no es necesario comprobar si cumple el criterio de aspiración.

6. **Estrategia back jump tracking**

La búsqueda no ha entrado en un ciclo, puesto que es la primera iteración. En esta iteración no se aplica la estrategia de back jump tracking.

7. **Criterio de parada**

Aún no se ha alcanzado el número máximo de iteraciones y hay movimientos disponibles, por lo que el algoritmo continúa.

2º iteración

1. **Actualización de los elementos**

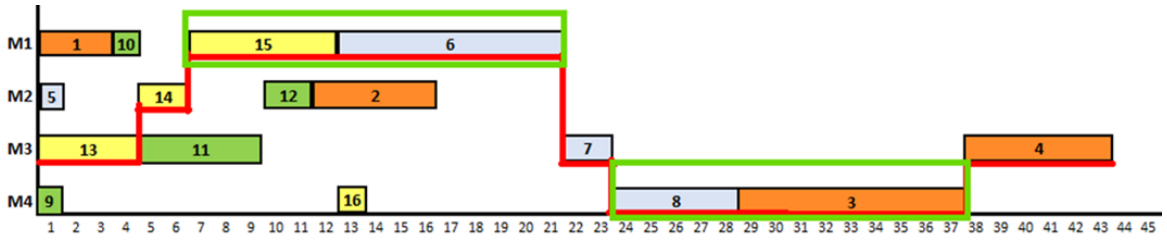
Se actualizan los diferentes elementos:

-Mejor valor encontrado = 31.

-Lista tabú = {3-8}

-Lista *back jump tracking*: como en la iteración anterior se ha mejorado el valor de la mejor solución encontrada (que al comienzo de la búsqueda era infinito y ahora es 31), se añaden a la lista *back jump tracking* los datos relativos a esa iteración:

-Solución de partida:



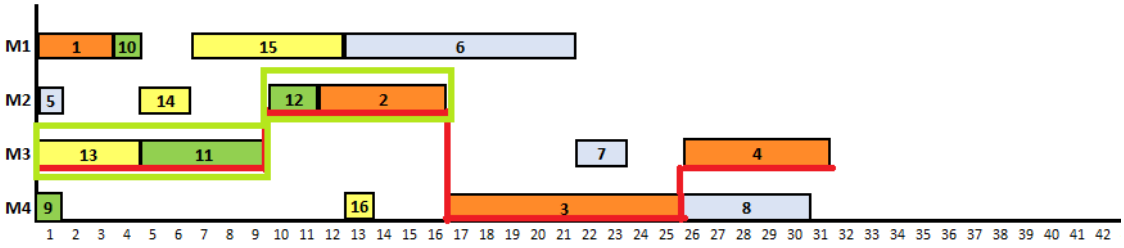
-Con *makespan* igual a 43.

-Lista tabú = { Ø }

-Movimiento efectuado = 8-3.

2. Solución inicial

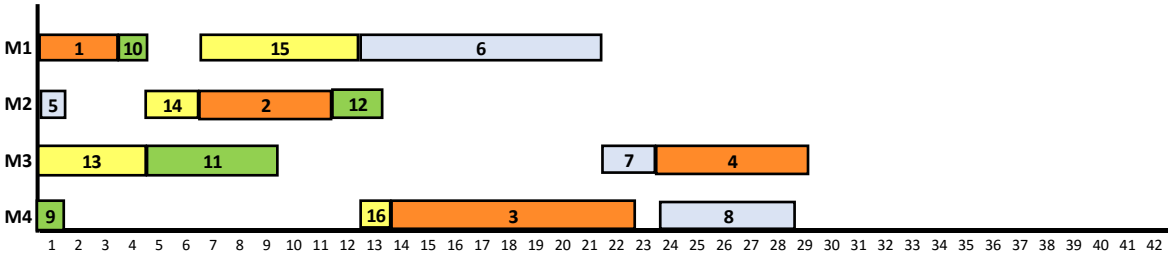
La solución inicial para esta iteración es la elegida en el paso anterior:



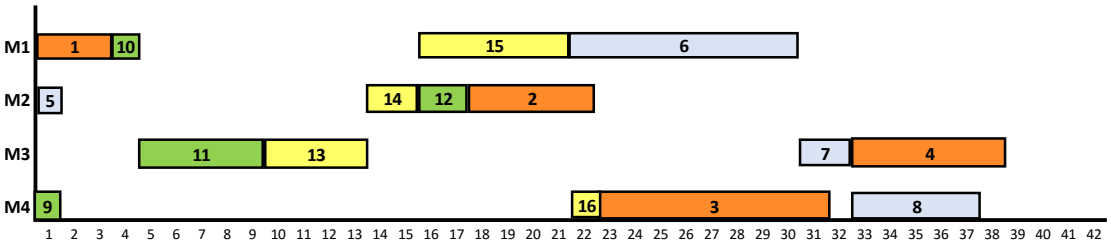
3. Generar entorno

La solución inicial tiene dos bloques con dos operaciones cada uno, por lo tanto existen dos movimientos disponibles, que serán:

* Intercambio de las operaciones 12-2, con makespan igual a 29:



* Intercambio de las operaciones 13-11, con makespan igual a 38:



La mejor solución es la generada a partir de intercambiar las operaciones 12-2.

El entorno no está vacío, por lo que no se ha llegado al óptimo.

4. **Comprobar si está en la lista tabú**

Lista tabú = {3-8}

No está en la lista, por lo que se puede elegir directamente

5. **Criterio de aspiración**

Al no aparecer en la lista tabú, no es necesario comprobar el criterio de aspiración.

6. **Estrategia *back jump tracking***

No se ha caído en un ciclo por lo que no se aplica la estrategia de intensificación.

7. **Criterio de parada**

Todavía hay disponibles movimientos para realizar y no se ha alcanzado el número máximo de iteraciones, por lo que no se cumplen los criterios de parada.

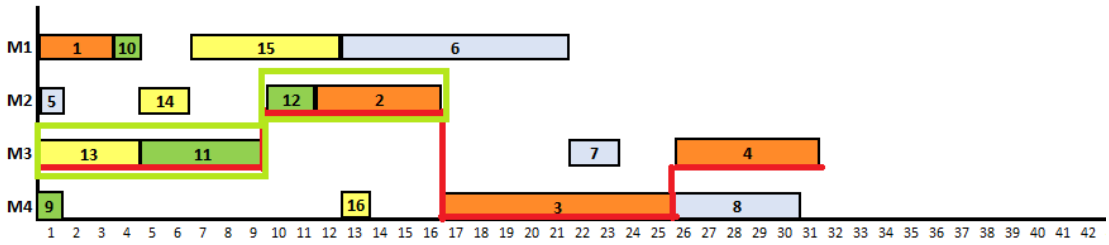
Para la siguiente iteración se tendría:

-Mejor valor encontrado = 29.

-Lista tabú = {3-8, 2-12}

-Lista *back jump tracking*: en la iteración anterior se ha mejorado el valor de la mejor solución encontrada (que al comienzo de la búsqueda era 31 y ahora es 29), se añaden a la lista *back jump tracking* los datos relativos a esa iteración:

-Solución de partida:



-Con *makespan* igual a 31.

-Lista tabú = { 3 − 8}

-Movimiento efectuado = 8-3.

4.5. Resultados obtenidos

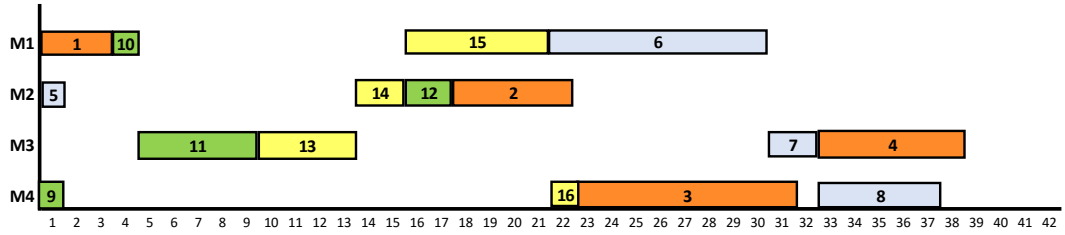
El algoritmo de búsqueda tabú como se ha mencionado al principio, se ha implementado en el software informático Matlab. La plataforma Matlab está especialmente diseñada para su utilización en problemas de ingeniería, debido a que entre sus funciones básicas incorpora la representación de datos y funciones, la implementación de algoritmos, además de trabajar fácilmente con matrices y funciones.

El código del programa desarrollado está incluido en el anexo al final de este trabajo, en el cual se explicará detalladamente el funcionamiento y la forma en la que se introducen los datos del problema.

El problema analizado ha sido el que se ha estudiado a lo largo de todo el trabajo, que es el siguiente:

Trabajo	A				B				C				D			
Operación	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Tiempo ejecución (p_i)	3	5	9	6	1	9	2	5	1	1	5	2	4	2	6	1
Máquina (M_i)	1	2	4	3	2	1	3	4	4	1	3	2	3	2	1	4

La solución inicial se ha obtenido aleatoriamente y es la siguiente, con un makespan de 38:



Los parámetros que hay que decidir a la hora de poner en marcha el programa, son la longitud de la lista tabú y el número de iteraciones máximo que ejecuta el programa. En este caso se ha decidido una lista tabú de longitud 8 y 30 iteraciones máximas.

En la primera iteración se obtiene un makespan de 29, en la segunda se reduce a 26, mientras que en la tercera iteración se alcanza el valor óptimo, que en este caso es 23. Se sabe que es así, porque 23 es la suma de las duraciones de todas las operaciones del primer trabajo (las operaciones 1, 2, 3 y 4), por tanto este valor no se puede reducir más. Además, como el camino crítico sólo estaría formado por las operaciones del primer trabajo, no existiría ningún bloque ni movimientos para realizar.

Tiene sentido que se alcance el óptimo en la tercera iteración puesto que el problema de estudio es pequeño, con cuatro trabajos y cuatro máquinas.

Capítulo 5

Conclusiones

Al inicio del trabajo se hablaba de la repercusión en trabajos, artículos y estudios que ha tenido el problema del job shop, esto no es de extrañar debido a las numerosas técnicas que se pueden aplicar para resolverlo y a su complejidad computacional, lo cual permite un amplio campo de investigación y mejora.

En el presente trabajo se han abordado tres métodos heurísticos diferentes, que no sólo se han estudiado de forma genérica, sino profundizando en su aplicación concreta al problema del *job shop.* Esto pone de manifiesto lo anteriormente comentado, que técnicas muy diferentes permiten resolver el JSSP.

Respecto a la implementación de la búsqueda tabú se han obtenido buenos resultados en problemas de pequeña dimensión, logrando valores muy cercanos al óptimo e incluso el óptimo. La mejora de esta implementación para obtener soluciones cuasi-óptimas en problemas de grandes dimensiones (20 trabajos por 20 máquinas por ejemplo) puede ser abordada en futuras investigaciones. Junto con esta mejora, se plantean además las siguientes líneas de investigación futura:

- En cuanto a los métodos aplicables en la resolución del problema del job shop, en el presente trabajo se han visto tres, sin embargo existen muchos más. Por tanto, se podría estudiar la aplicación de otros métodos tales como el recocido simulado, la metaheurística GRASP o el algortimo de colonia de hormigas entre muchos otros.

- En cuanto a la implementación de la búsqueda tabú, se ha visto que aun queda mucho campo de mejora. Algunas estrategias que se podrían implementar en un futuro son la estrategia de back jump tracking y un algoritmo que obtenga una solución inicial de calidad.

- Otra línea futura de investigación sería la implementación de algoritmo genéticos y algoritmo de colonia de abejas en un entorno informáticos, al igual que se ha hecho con la búsqueda tabú.

Así, animando al lector a continuar con estas líneas de investigación propuestas, ponemos punto y final a este proyecto.

BIBLIOGRAFÍA

-Por riguroso orden alfabético.-

(1) J. ADAMS, J. E. BALAS, D. ZAWACK (1988), *The shifting bottleneck procedure for job shop scheduling*, Management Sci. 34, 391-40.

(2) R. M. AIEX, S. BINATO, M. G. C. RESENDE (2003), *Parallel GRASP with Path-Relinking for Job Shop Scheduling*. Parallel Computing, 29. pp. 393-430.

(3) S. B. AKERS, J. FRIEDMAN, *A non-numerical approach to production scheduling problems*, Oper. Res. 3 (1955) 429 – 442.

(4) A. D. AGUILAR (2014), *Un algoritmo basado en la colonia artifcial de abejas con búsqueda local para resolver problemas de optimización*, Tesis, Universidad Veracruzana.

(5) Algorítmica (2013): Búsqueda Tabú. Recuperado en junio 2016, de: *<http://sci2s.ugr.es/sites/default/files/files/Teaching/GraduatesCourses/Algoritmica/Tema04-BusquedaTabu-12-13.pdf >*

(6) E. J. ANDERSON, C. A. GLASS, C. N. POTTS (1995), *Local search in combinatorial optimization: applications in machine scheduling*, Research Report No. OR56, University of Southampton.

(7) D. APPLEGATE, W. COOK (1991), *A computational study of the Job-Shop Scheduling Problem*, ORSA Journal on Computing, Vol.3, Nº2, 149-156.

(8) J. A. ARAÚZO (2015), *Introducción a la Dirección de Operaciones*, Asignatura Dirección de Operaciones, Universidad de Valladolid.

(9) J. A. ARAÚZO (2016), *Programación de Talleres*, Asignatura Dirección de Operaciones, Universidad de Valladolid.

(10) J. ARRANZ DE LA PEÑA, A. PARRA TRUYOL, Algoritmos genéticos, Universidad Carlos III.

(11) S. ASHOUR, S.R. HIREMATH, *A branch-and-bound approach to the job-shop scheduling problem,* Internat. J. Prod. Res. 11, 1977, 47-58.

(12) E. BALAS (1969)*, Machine sequencing via disjunctive graphs: An implicit enumeration algorithm,* Oper. Res. 17, 941-957.

(13) E. BALAS, J. K. LENSTRA, A. VAZACOPOULOS (1995), *One machine scheduling with delayed precedence constrains*, Management Sci. 41, 94-109.

(14) E. BALAS, A. VAZACOPOULOS (1995), *Guided local search with shifting bottleneck for job shop scheduling.*

(15) *I. BARBA, Algoritmos de planificación basados en restricciones para la sustitución de componentes defectuosos, Tesis doctoral. Departamento de Lenguajes y Sistemas informáticos, Universidad de Sevilla.*

(16) S. BINATO, W. J. HERY, D. M. LOEWENSTERN, M. G. C. RESENDE (2002), *A GRASP for Job Shop Scheduling*. In: Essays and Surveys in Metaheuristics, Ribeiro, Celso C., Hansen, Pierre (Eds.), Kluwer Academic Publishers.

(17) J. BLAZEWICZ, K. H. ECKER, E. PESCH, G. SCMIDT, J. WEGLARZ (1996), Scheduling Computer and Manufacturing Processes, Springer.

(18) G. H. BROOKS, C. R. WHITE (1965), *An algorithm for finding optimal or near-optimal solutions to the production scheduling problem*, J. Industrial Eng. 16, , 34-40.

(19) P. BRUCKER (1988), *An efficient algorithm for the job-shop problem with two jobs. Computing 40, 353 – 359.*

(20) P. BRUCKER, B. JURISCH, B. SIEVERS (1994), *A Branch and Bound Algorithm for Job-Shop Scheduling Problem*, Discrete Applied Mathematics, Vol 49, pp. 105-127.

(21) J. CARLIER, E. PINSON*, An algorithm for solving the job.shop problem*, Management Sci. 35, 1989, 164-176

(22) D. CORTÉS RIVERA (2004), *Un Sistema Inmune Artificial para resolver el problema del Job Shop Scheduling,* Tesis, CINVESTAV, p.19.

(23) J.M. CHARLTON, C.C. DEATH, *A generalized machine scheduling algorithm*, Per. Res. Quart. 21, 1970, 127-134.

(24) E. CUEVAS (2015), *El algoritmo "Artificial Bee Colony" (ABC) y su uso en el Procesamiento digital de Imágenes*. Journal Iberamia. Inteligencia Artificial 18(55), 50-68.

(25) S. DAUZERE-PERES, J.B. LASSERRE (1993), *A modified shifting bottleneck procedure for job shop scheduling*, Internat. J. Prod. Res. 31, 923-932.

(26) Decide (2014), *Adaptación del algoritmo de colonia de abejas al cálculo de rutas*. Recuperado en junio 2016, de:

<*http://www.decidesoluciones.es/adaptacion-del-algoritmo-de-colonia-de-abejas-al-calculo-de-rutas/* >

(27) M. DELL' AMICO, M. TRUBIAN (1993), *Applying tabu-search to the job shop scheduling problem*, 231,252.

(28) D. Z. DU, P. M. PARLADOS (1998), *Handbook of Combinatorial Optimization, Volumen 3*. Kluwer Academic Publishers.

(29) H.FISHER, G.L. THOMPSON (1963), *Probabilistic learning combinations of local job-shop scheduling rules,* in J. F. Muth, G. L. Thompson (eds.), Industrial Scheduling, Prentice Hall, Englewoods Cliffs.

(30) M. L. FISHER (1973), *Optimal solution of scheduling problems using Lagrange multipliers*; Part I, Oper. Res. 21, 1114-1127.

(31) M. FLORIAN, P. TRÉPANT, G. MCMAHON, *An implicit enumeration algorithm for the machine sequencing problem*, Management Sci. 17,1971, B782-B792.

(32) C. A. GLASS, C. N. POTTS, P. SHADE (1992), *Genetic algorithms and neighbourhood search for scheduling unrelated parallel machines*, Working paper No. OR47, University of Southampton.

(33) F. GLOVER (1987), *Tabu Search Methods in Artificial Intelligence and Operations Research*, ORSA Artificial Intelligence, Vol. 1, No. 2, 6.

(34) F. GLOVER, M. LAGUNA (1995), *Moderns Heuristic Techniques for Combinational Problems*, McGraw Hill Book Co.

(35) F. GLOVER, R. HÜBSCHER (1991), *Binpacking with a tabu search*, Technical Report, University of Colorado.

(36) M. A. GONZÁLEZ FERNÁNDEZ (2011), *Soluciones Metaheurísticas al "Job-Shop Scheduling Problem with Sequence-Dependent Setup Times"*, Tesis doctoral, Universidad de Oviedo, p.1.

(37) H. H. GREENBERG (1968), *A branch and bound solution to the general scheduling problem*. Oper. Res.16, 353-361.

(38) P. HANSEN (1986), *The steepest ascent mildest descent heuristic for combinatorial programming*, Congress on Numerical Methods in Combinatorial Optimization.

(39) J. HOLLAND (1975), *Adaptation in Natural and Artificial System.*

(40) L. P. INGOLOTTI HETTER (2007), *Modelización y métodos para la optimización y eficiencia de la programación de horarios ferroviarios,* Tesis doctoral, Universidad de Valencia.

(41) J.R. JACKSON (1956) , An extension of Johnson's Result on Job Lot Scheduling, Naval Research Logistics Quaterly, 3, pp. 201-203.

(42) D. S. JOHNSON M. R. GAREY (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Series of Books in the Mathematical Sciences.

(43) S.M. JOHNSON (1954), *Optimal two and three stage production schedule with setup times included*, Naval Research Logistics Quaterly, 61-67.

(44) D. KARABOGA (2005). *An idea based on honey bee swarm for numerical optimization*, Technical Report TR06, Erciyes University, Engineering Faculty, Computer Engineering Department.

(45) G. MAILING (2003), *Algoritmos heurísticos y el problema de job shop scheduling*, Facultad de Ciencias Exactas y Naturales Universidad de Buenos Aires.

(46) A. MATEOS ANDALUZ, *Algoritmos evolutivos y algoritmos genéticos,* Universidad Carlos III de Madrid.

(47) H. MATSUO, C. J. SUH, R. S. SULLIVAN (1988*), A controlled search simulated annealing method for the general job shop scheduling problem*, working paper 03-04-88, University of Texas Austin.

(48) D. C. MATTFELD (1995), *Evolutionary search and the job shop*, Ph. D. thesis, University of Bremen.

(49) J. F. MUTH, G. L. THOMPSON (1963), *Industrial Scheduling, Prentice-Hall, Englewood Cliffs*, NJ.

(50) E. NOWICKI, C. SMUTNICKI (1996), A Fast Taboo Search Algorithm for the Job Shop Problem, Management Science, Vol. 42, No. 6, 797-813.

(51) OBS Business School (2014), *¿Qué es un diagrama de Gantt y para qué sirve? .* Recuperado en junio 2016, de: <http://www.obs-edu.com/blog-project-management/diagramas-de-gantt/que-es-un-diagrama-de-gantt-y-para-que-sirve/>

(52) J. C. OSORIO, O. E. CASTRILLÓN, J. A. TORO, J. P. OREJUELA (2008), *Modelo de programación jerárquica de la producción en un Job shop flexible con*

interrupciones y tiempos de alistamiento dependientes de la secuencia, Revista Ingeniería e Investigación vol. 28 no. 2, 72-79.

(53) V. PEÑA, L. ZUMELZU (2006), *Estado del Arte del Job Shop Scheduling Problem*, Departamento de Informática, Universidad Técnica Federico Santa María, Chile.

(54) M. E. PÉREZ, Web Personal (2011), *Job Shop Scheduling Problem, Benchmarking Instance*s. Recuperado en junio de 2016, de: <*http://www.eii.uva.es/elena/JSSP/InstancesJSSP.htm*>

(55) M. E. PÉREZ (2010), *Guía para recién llegados a los Algoritmos Genéticos*, Universidad de Valladolid.

(56) M. E. PÉREZ, A. GENTO, A. REDONDE, R. DEL OLMO, J. J. BENITO, J. A. ARAÚZO (2010), *Herramientas modernas de optimización*, Universidad de Valladolid.

(57) M. T. PÉREZ RODRÍGUEZ (2015), *Recocido Simulado*, Asignatura Métodos Matemáticos, Universidad de Valladolid.

(58) B. PÉREZ DE VARGAS (2015), *Resolución del Problema del Viajante de Comercio (TSP) y su variante con Ventanas de Tiempo (TSPTW) usando métodos heurísticos de búsqueda local*, TFG, Universidad de Valladolid, 19-25.

(59) M. L. PINEDO (2005), *Planning and Scheduling in Manufacturing and Services*, Springer, 3-8.

(60) M. G. RAVETTI, G. R. MATEUS, P. L. ROCHA, P. M. A. PARDALOS (2007), *Scheduling Problem with Unrelated Parallel Machines and Sequence Dependent Setups*. International Journal of Operational Research 2, pp. 380-399.

(61) H. C. REYES, C*amino crítico, introducción teórica*, Cátedra de organización de la producción.

(62) B. ROY, B, B. SUSSMAN (1964), *Les problemes d'ordonnancement avec constraintes disjonctives*: SEMA, Note D.S., Nº. 9, Paris.

(63) S. M. SAIT, H. YOUSSEF (1999), Iterative Computer Algorithms with Applications in Engineering, IEEE Computer Society Press, California, USA.

(64) Y.N. SOTSKOV (1985), *Optimal scheduling two jobs with regular criterion*. Institute of Engineering Cy-bernetics, Minsk, Belarus, 86-95.

(65) Y.N. SOTSKOV (1991), *The complexity of shop-scheduling problems with two or three jobs*, European J. Oper. Res.53, 326 - 336.

(66) W. SZWARC (1960*), Solution of the Akers-Friedman scheduling problem*. Oper. Res.8/6, 782 – 788.

(67) E. TAILLARD (1989), Benchmarks for basic scheduling problems. Recuperado en junio 2016, de: <*http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/ordonnancement.html*>

(68) Todo sobre ciencia, Teoría evolución de Darwin. Recuperado en junio de 2016 de: <*http://www.allaboutscience.org/spanish/teoria-de-la-evolucion-de-darwin.htm*>

(69) R. J. P. VAESSENS, E. H. L. AARTS, J. K. LENSTRA (1995), *Job shop scheduling by local search*, Working paper, University of Technology, Eindhoven.

(70) P. J. M. VAN LAARHOVEN, E. H. L. AARTS, J. K. LENSTRA (1992), *Job shop scheduling by simulated annealing,* Oper. Res. 40, 113-125.

(71) Wikipedia (2016), Inteligencia de enjambre. Recuperado en junio 2016, de: <*https://es.wikipedia.org/wiki/Inteligencia_de_enjambre* >

(72) Wikipedia (2016), *Teoría de la complejidad computacional*. Recuperado en junio de 2016, de: <*https://es.wikipedia.org/wiki/Teor%C3%ADa_de_la_complejidad_computacional#Algoritmos_de_tiempo_polin.C3.B3mico_y_problemas_intratables*>

(73) D. P. WILLIAMSON, L. A. HALL, J. A. HOOGEVEEN, C. A. J. HURKENS, J. K. LENSTRA, S. SEVAST'JANOV, D. B. SCHMOYS (1997*), Short Shop Schedules*, Operations Research, March - April, 288-294.

(74) T. YAMADA, R. NAKANO (1992), *A genetic algorithm applicable to large-scale job-shop problems*, in: R. Männer, B. Manderick (eds.), Parallel Problem Solving from Nature 2.

(75) T. YAMADA, R. NAKANO (1997), *Genetic algorithms in engineering systems*, A.M.S. Zalzala and P.J. Fleming, 134–160.

(76) R. ZHANG, S. SONG, C. WU (2013), *A hybrid artificial bee colony algorithm for the job shop scheduling problem*, Int. J. Production Economics, 167-178.

Anexo I. Código del programa

Como ya se ha comentado anteriormente, como parte del presente trabajo se ha procedido a la implementación del algoritmo de búsqueda tabú. Dicha implementación se ha desarrollado mediante la herramienta de software matemático Matlab, el cual ofrece un entorno de desarrollo integrado con un lenguaje de programación propio (lenguaje M).

A continuación se explicará el funcionamiento del programa además de exponer el código con sus correspondientes funciones.

Recordamos que un problema *job shop* se representa mediante un conjunto de operaciones, en el que cada una de ellas tiene asociado un trabajo al que pertenece, una máquina en la que ser procesada y un tiempo de ejecución. Además las operaciones pertenecientes a un mismo trabajo tienen un orden establecido en el que ser procesadas. El objetivo es hallar la secuencia de máquinas que cumple un determinado objetivo, que en este caso será minimizar el tiempo total del proceso, es decir, minimizar el *makespan*.

El programa trabajará con la idea del problema representado mediante un grafo, en el que cada nodo simboliza una operación. Para ello será necesario añadir dos nodos ficticios, un nodo fuente y un nodo sumidero que representan el inicio y el fin del proceso respectivamente.

Dicho grafo se implementa en Matlab mediante una matriz de adyacencia en la que cada fila y cada columna están asignadas a una operación del problema. Si existe una arista entre el nodo a y el nodo b, el elemento (a, b) de la matriz (fila a y columna b) será 1, en caso contrario, será 0.

Además de la matriz de adyacencia, un grafo también se representa con la matriz de pesos, en la cual, en vez de poner un 1 en caso de que exista una arista, se asigna el peso de dicha arista. En caso de que una arista entre dos nodos no exista, se puede representar en la matriz de pesos mediante 0, infinito o menos infinito, dependiendo de las necesidades del problema. Para este caso se ha utilizado el valor menos infinito.

El enunciado del problema se introduce en el programa mediante una matriz de datos. Cada columna de la matriz de datos está referida a una operación del problema, teniendo la matriz tantas columnas como operaciones haya. Las filas de la matriz indican lo siguiente:

1º Fila: número del trabajo al que pertenece la operación.

2º Fila: número de máquina en la que se debe procesar.

3º Fila: tiempo de ejecución.

4º Fila: número que indica la posición en relación a la secuencia del trabajo al que pertenece.

5º Fila: número que indica la posición en relación a la secuencia de la máquina a la que pertenece.

Las cuatro primeras filas son datos fijos del problema, en cambio la última fila variará a lo largo de la búsqueda e indica las soluciones factibles.

La solución inicial a partir de la cual el algoritmo comienza, será introducida por el usuario en esa última fila, y podrá ser generada bien de forma aleatoria o bien mediante alguna técnica de optimización. Este proceso de generación no está incluido en el programa y como ya se ha dicho, será el usuario el encargado de introducir una solución inicial en la matriz de datos.

Los parámetros que necesita el programa son el número máximo de iteraciones a partir del cual se detiene y la longitud de la lista tabú. Además el usuario deberá especificar el número de máquinas y el número de operaciones, que se puede deducir de la matriz de datos.

Para simplificar, el programa está diseñado para problemas en los que los trabajos tienen tantas operaciones como máquinas haya. Por ejemplo, si se tienen tres máquinas y cinco trabajos, cada trabajo estará formado por tres operaciones, procesadas cada una de ellas en una máquina diferente.

Es necesario señalar que aunque la estrategia de back tracking se ha explicado anteriormente como parte del algoritmo, no ha sido implementada en el presente programa.

Una vez introducido la forma de representar el programa, se expone el código de programación con todas las funciones correspondientes.

PROGRAMA

```
function [S, lmin] = programa( MD, n, M, ltabu, ITER)
%Función que calcula la solución que minimiza el makespan (o una
muy próxima) para un problema dado tipo job shop.

%ENTRADA: MD: matriz de datos
          n: número de operaciones
          M: número de máquinas
          ltabú: longitud de la lista tabú
          ITER: número máximo de iteraciones

%SALIDA: S: vector de la mejor solución encontrada
         lmin: mínimo makespan encontrado


nno=n+2; %nno = Numero de nodos

niter=0; %variable que indica el número de iteración

%INICIALIZACIÓN LISTA TABÚ (La lista tabú tiene tantas filas como
su longitud y dos columnas en las que se indican las operaciones
intercambiadas)
for i=1:ltabu
    tabu(i,1)=0;
    tabu(i,2)=0;
end

%SOLUCIÓN INICIAL
%transforma la matriz de datos en la matriz de adyacencia
[MAs] = matDmatA(MD, n, M);
%transforma la matriz de adyacencia en la matriz de pesos
[MPs] = matAmatP(MAs, MD, n);
%determina el makespan y el vector de precedencias de un programa
[d, Vpred] = AlgCCritico(MPs,nno);
%Determina el camino crítico
[ Vcc ] = VecCC( Vpred, nno );
%Encuentra los bloques y realiza los movimientos
 [ MSMs, MO] = BloquesyMvtos( MD,Vcc,n );

lmin=d;
S=MD(5,:);

%el número de programas nuevos (movimientos diferentes) es el nº
de filas de la matriz MSMs
limite=size(MSMs,1);

dminimo=inf; %variable para ver cuál es el mejor movimiento

%ANALIZAMOS LOS MOVIMIENTOS DE LA SOLUCIÓN INICIAL
for i=1:limite
    MDNuevo=[MD(1:4,:);MSMs(i,:)]; %MDnuevo: matriz de datos nueva
con la secuencia de máquinas del mvto correspondiente
    [MAs] = matDmatA(MDNuevo, n, M);
    [MPs] = matAmatP(MAs, MD, n);
    [d, Vpred] = AlgCCritico(MPs,nno);
```

```
    distancia(i)=d;

    if d<dminimo
       dminimo=d; %Comprobar cuál es el mejor de los movimientos.
       indice=i;
    end
end

if lmin>dminimo %si mejora al mínimo mejor encontrado, se
sustituye
    lmin=dminimo;
    S=MSMs(indice,:);
end

%REPETICIÓN CON LA SECUENCIA NUEVA
MD1=[MD(1:4,:);MSMs(indice,:)]; %MD1: matriz de datos nueva
%Se añade el movimiento realizado al final de la lista tabú
tabu(ltabu-1,:)=MO(indice,:);
aux=MO(indice,1);
MO(indice,1)=MO(indice,2);
MO(indice,2)=aux;
tabu(ltabu,:)=MO(indice,:);

%PARA LAS SIGUIENTES ITERACIONES
while niter<ITER
    %vaciar MSMs y MO
    MSMs(:,:)=[];
    MO(:,:)=[];

    %crear los nuevos mvtos a partir de MD1
    [MAs] = matDmatA(MD1, n, M);
    [MPs] = matAmatP(MAs, MD1, n);
    [d, Vpred] = AlgCCritico(MPs,nno);
    [ Vcc ] = VecCC( Vpred, nno );
    [ MSMs, MO] = BloquesyMvtos( MD1,Vcc,n );

    if MSMs==0 %en caso de que no haya movimeintos, la función de
BloquesyMvtos, devuelve MSMs=0
        break; %eso quiere decir que se ha alcanzado el óptimo y
se sale del programa
    end

    %analizar los movimientos obtenidos de MD1
    limite=size(MSMs,1);
    dminimo=inf;
    for i=1:limite
        MDNuevo=[MD1(1:4,:);MSMs(i,:)];
        [MAs] = matDmatA(MDNuevo, n, M);
        [MPs] = matAmatP(MAs, MD, n);
        [d, Vpred] = AlgCCritico(MPs,nno);
        distancia(i)=d;
        if d<dminimo
           dminimo=d;
           indice=i;
           MD1=MDNuevo;
        end
    end
```

```
   %sacamos de la matriz de movimientos el elegido y se guardamos
en la variable movimiento
   movimiento=MO(indice,:);
   estabu=0;%varible para indicar si un mvto está en la lista tabú
   for j=1:ltabu
         if isequal(movimiento,tabu(j,:)); %si está en la lista
tabú
             estabu=1; %la variable estabú se marca con 1
         end
   end

    %SI ESTÁ EN LA LISTA TABÚ
   if estabu==1

      %CRITERIO DE ASPIRACIÓN
         if dminimo<lmin %si lo cumple
              lmin=dminimo;
              MD1=[MD1(1:4,:);MSMs(indice,:)];
             S=MSMs(indice,:);
         else %si no cumple el criterio de aspiración
              regenerar=0;
              seguridad=0;
              while regenerar==0 & seguridad<ltabu
                    for mov=2:ltabu
                        tabu(mov-1,:)=tabu(mov,:);
                    end
                    tabu(ltabu,:)=tabu(ltabu-1,:);

                    for j=1:ltabu
                         if isequal(movimiento,tabu(j,:));
                                 regenerar=0;
                         else
                             regenerar=1;%si el mvto ha salido de
la lista tabú, ya se puede elegir-> se  sale del while
                         end
                     end
                     seguridad=seguridad+1;
              end

            MD1=[MD1(1:4,:);MSMs(índice
            if lmin>dminimo
                  lmin=dminimo;
                  S=MSMs(indice,:);
            end
            for l=1:ltabu-2
                tabu(l,:)=tabu(l+2,:);
            end
            tabu(ltabu-1,:)=MO(indice,:);
            aux=MO(indice,1);
            MO(indice,1)=MO(indice,2);
            MO(indice,2)=aux;
            tabu(ltabu,:)=MO(indice,:);
         end


    %SI NO ESTÁ EN LA LISTA TABÚ
```

```
   else
            MD1=[MD1(1:4,:);MSMs(indice,:)];
            if lmin>dminimo
                  lmin=dminimo;
                  S=MSMs(indice,:);
            end
            for l=1:ltabu-2
                tabu(l,:)=tabu(l+2,:);
            end
            tabu(ltabu-1,:)=MO(indice,:);
            aux=MO(indice,1);
            MO(indice,1)=MO(indice,2);
            MO(indice,2)=aux;
            tabu(ltabu,:)=MO(indice,:);
        end
    niter=niter+1;
end

end
```

FUNCIÓN QUE TRANSFORMA LA MATRIZ DE DATOS EN LA MATRIZ DE ADYACENCIA

```
function [MAs] = matDmatA( MD,n,M)
%Transforma la matriz de datos en la matriz de adyacencia
%ENTRADA: MD: Matriz de datos
          n: número de operaciones
          M: número de máquinas

%SALIDA: Mas: Matriz de adyacencia

%INICIALIZAR matriz con todos sus valores como 0
MA(n+2,n+2)=0;

%1º FILA (arcos desde el nodo fuente->valen 1 para la primera
operación de cada trabajo
for j=1:n
 if MD(4,j)==1
     MA(1,j+1)=1;
 end
end

%SECUENCIA DE TRABAJOS
for i=2:n+1
    for j=2:n
        if i==j
            if MD(4,j)~=1
                MA(i,j+1)=1;
            else
                MA(i,n+2)=1;
            end
        end
    end
end
MA(n+1,n+2)=1;%como se toma de referencia el siguiente trabajo,
```

```
para el último no nos sirve, pero su última operación siempre va
al último nodo

%SECUENCIA DE MÁQUINAS
Mmaq(M,n/M)=0; %Mmaq: matriz auxiliar. Las filas indican el n° de
la máq, y en cada fila se ponen las operaciones de esa máq en el
orden de secuenciación
for m=1:M
    contm=1; %contador que marca el orden de secuenciación
    while Mmaq(m,n/M)==0
        for j=1:n
            if MD(2,j)==m
                if MD(5,j)==contm
                   Mmaq(m,contm)=j
                   contm=contm+1;
                end
            end
        end
    end
end

%transportamos los datos de la matriz auxiliar a la MA
for i=1:M
    for j=1:((n/M)-1)
        MA(Mmaq(i,j)+1,Mmaq(i,j+1)+1)=1;
    end
end

MAs=MA;
end
```

FUNCIÓN QUE TRANSFORMA LA MATRIZ DE ADYACENCIA EN LA MATRIZ DE PESOS

```
function [ MPs ] = matAmatP(MA, MD, n )
%Transforma la matriz de adyacencia en la matriz de datos
%ENTRADA: MA: matriz adyacencia
          MD: matriz datos
          n: número de operaciones

%SALIDA: MPs: matriz de pesos

nul=-inf;

%Inicialización de la matriz de datos como menos infinito
for i=1:n+2
    for j=1:n+2
        if MA(i,j)==0
            MP(i,j)=nul;
        end
    end
end

%1° FILA
for j=1:n+2
```

```
    if MA(1,j)==1
        MP(1,j)=0;
    end
end

%DIAGONAL es todo 0
for i=1:n+2
    for j=1:n+2
        if i==j
            MP(i,j)=0;
        end
    end
end
%DURACIÓN OPERACIONES
for i=2:n+2
    for j=2:n+2
        if MA(i,j)==1
            MP(i,j)=MD(3,i-1);
        end
    end
end

MPs=MP;

end
```

FUNCIÓN QUE CALCULA EL VECTOR DE PRECEDENCIAS PARA EL CAMINO CRÍTICO Y EL MAKESPAN

```
function [dist,  Vpred] = AlgCCritico(MP,n )
% función que calcula el vector de precedencias para el camino
crítico de un grafo y el valor del makespan

%ENTRADA: MP: matriz de pesos
          n: número de nodos (n° operaciones más 2 (nodo fuente y
          nodo      sumidero)

%SALIDA: dist: makespan del programa
         Vpred: vector de precedencias entre nodos

d(1)=0;
pred(1)=0;
L(1)=1;

for i=2:n
    L(i)=0;)
end

for j=2:n
d(j)=-inf;
end

cont =1; %Variable que indica si la lista está vacía (0) o no (1)
while cont==1
    for i=1:n
```

```
        if L(i)==1;%si el nodo está en la lista, trabajamos con él
            L(i)=0; %y le sacamos de la lista
            for j=2:n
                if d(j)<d(i)+MP(i,j
                    d(j)=d(i)+MP(i,j);
                    pred(j)=i;

                    if L(j)==0
                        L(j)=1;
                    end
                end
            end
        end
    end
    cont=0;
    for j=1:n
        if L(j)==1
            cont=1;
        end
    end
end

dist=d(n);
Vpred=pred;

end
```

FUNCIÓN QUE TRANSFORMA EL VECTOR DE PRECEDENCIAS EN EL CAMINO CRÍTICO

```
function [ Vcc ] = VecCC( Vpred, n )
%A partir del vector Vpred de precedencias, obtenemos el vector
del camino crítico (Vcc).
%El vector de precedencias tiene como longitud el número de nodos
del grafo. El número de una posición determinada indica el número
del nodo precedente en el camino crítico al nodo de esa posición.

%ENTRADA: Vpred: vector de precedencias
          n: número de nodos

%SALIDA: Vcc: vector que indica el camino crítico

VcAr(1)=n; Variable auxiliar que representa un vector del camino
crítico al revés (del último nodo al primero)

 i=2;
while VcAr(i-1)~=1
     VcAr(i)=Vpred(VcAr(i-1));
     i=i+1;
 end

l=length(VcAr);

%Obtener el vector del camino crítico a partir del vector auxiliar
j=l;
```

```
i=1;
   while (i<=l)&&(j>=1)
    Vc(i)=VcAr(j);
    j=j-1;
    i=i+1;
   end

 Vcc=Vc;
end
```

FUNCIÓN QUE OBTIENE LAS NUEVAS SECUENCIAS DE LAS MÁQUINAS A PARTIR DE REALIZAR LOS MOVIMIENTOS EN LOS BLOQUES

```
function [ MSMs, MO] = BloquesyMvtos( MD,Vcc,n )
%A partir del camino crítico, determina los bloques y efectúa los
movimientos, obteniendo como resultado la nueva secuencia de
máquinas.

%ENTRADA: MD: matriz de datos
          Vcc: vector del camino crítico.
          n: número de operaciones

%SALIDA: MSMs: matriz de las nuevas secuencias en la máquina (con
           los movimientos ya hechos)
         MO: matriz que almacena los movimientos efectuados. Tiene
           tantas filas como movimientos hechos y tiene dos
           columnas las cuales indican las dos operaciones
           intercambiadas.

contbloque=0; %variable que cuenta el número de bloques
contop=1; %variable que cuenta el n° de operaciones de cada bloque
l=length(Vcc);

%inicialización del Vector de secuencia de máquinas como el de la
solución inicial
for i=1:n
    VSM(i)=MD(5,i);
    VSMtemp(i)=MD(5,i); %esta variable guarda siempre la
secuencias de máquinas original
end


for i=2:l-2
    if MD(2,Vcc(i)-1)==MD(2,Vcc(i+1)-1) %si las máquinas son
iguales
        contop=contop+1;
        contopaux=contop;

    else %si las máquinas no son iguales
        if contop>=2 %si hay dos o más operaciones->es un bloque
            contbloque=contbloque+1;
            contopaux=contop;
```

```
for k=1:contop
    VB(k)=Vcc(i-contop+k); %vector con todas las
                           operaciones del bloque
end

%1° bloque
if contbloque==1
    aux=VSM(VB(contop)-1);
    VSM(VB(contop)-1)=VSM(VB(contop-1)-1);
    VSM(VB(contop-1)-1)=aux;
    MO(contbloque,1)=VB(contop-1);
    MO(contbloque,2)=VB(contop);


  %una vez hecho el intercambio, se descarga el
  vector en la fila correspondiente en la matriz
  de secuencia de máquinas
    for u=1:n
        MSM(contbloque,u)=VSM(u);
    end

for x=1:n %inicialización del Vector de secuencia
        de máquinas como el de la solución inicial
     VSM(x)=VSMtemp(x);
end

%bloques intermedios
else
    if contop==2 %si el bloque tiene 2 operaciones
       aux=VSM(VB(contop)-1);
        VSM(VB(contop)-1)=VSM(VB(contop-1)-1);
      VSM(VB(contop-1)-1)=aux;
       MO(contbloque,1)=VB(contop-1);
        MO(contbloque,2)=VB(contop);
        for u=1:n
              MSM(contbloque,u)=VSM(u);
        end

        for x=1:n
              VSM(x)=VSMtemp(x);
         end

    else %si tiene más de 2 operaciones
         %se intercambian las últimas
        aux=VSM(VB(2)-1);
        VSM(VB(2)-1)=VSM(VB(1)-1);
        VSM(VB(1)-1)=aux;
         MO(contbloque,1)=VB(1);
        MO(contbloque,2)=VB(2);

        for u=1:n
              MSM(contbloque,u)=VSM(u);
        end
        for x=1:n
              VSM(x)=VSMtemp(x);
        end
       % se %intercambian las primeras
```

```
                        aux=VSM(VB(contop)-1);
                       VSM(VB(contop)-1)=VSM(VB(contop-1)-1);
                        VSM(VB(contop-1)-1)=aux;
                        MO(contbloque,1)=VB(contop-1);
                        MO(contbloque,2)=VB(contop);
                        for u=1:n
                              MSM(contbloque,u)=VSM(u);
                        end

                        for x=1:n
                               VSM(x)=VSMtemp(x);
                         end
                    end
                end
                contop=1; %inicialización de ambas variables
                contopaux=1;
        else
            contop=1;
            contopaux=1;
        end
    end
end

%para el último bloque
if contopaux>=2 %si hay más de dos operaciones
      contbloque=contbloque+1;
                    for k=1:contopaux
                        VB(k)=Vcc(i-contopaux+1+k);
                    end
                    aux=VSM(VB(2)-1);%se intercambian las primeras
                    VSM(VB(2)-1)=VSM(VB(1)-1);
                    VSM(VB(1)-1)=aux;
                    MO(contbloque,1)=VB(1);
                    MO(contbloque,2)=VB(2);

                    for u=1:n
                         MSM(contbloque,u)=VSM(u);
                    end

end
if contbloque>0
    MSMs=MSM;
    MOs=MO;
else
   MSMs=0;
   MO=0;
end
end
```