View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UCrea

Facultad de Ciencias

UN ALGORITMO MEMÉTICO PARA EL JOB SHOP SCHEDULING PROBLEM
(A memetic algorithm for the job shop scheduling problem)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Pablo García Gómez

Director: Inés González Rodríguez

Junio - 2019

Resumen

Planificar tareas es un proceso que los humanos realizamos diariamente, de forma inconsciente, para llevar a cabo distintas labores. Sin embargo, en el ámbito industrial, la planificación de tareas no puede realizarse sin un análisis previo, pues las decisiones afectarán a los resultados y estos a las ganancias económicas.

El *job shop scheduling problem* es un problema clásico que trata de modelar de forma genérica aquellas situaciones en las que hay que planificar la ejecución de una serie de operaciones sobre un conjunto finito de recursos.

Desafortunadamente, la complejidad del problema es *NP-hard*, lo que hace inviable el uso de técnicas que aseguren encontrar una solución óptima. Es en esta situación donde cobran relevancia las técnicas metaheurísticas, porque aunque no dan garantías acerca de la optimalidad de la solución retornada, permiten encontrar soluciones suficientemente buenas en un tiempo razonable. Además, los algoritmos metaheurísticos son especialmente interesantes porque su funcionamiento es independiente del problema que tratan de resolver, es decir, pueden ser adaptados con facilidad para resolver una amplia variedad de problemas. Por esta razón, son una de las áreas de investigación más activas en el ámbito de los problemas de optimización.

En este trabajo se propone un algoritmo memético, un tipo concreto de metaheurística que explota las sinergias entre dos estrategías metaheurísticas distintas, un algoritmo evolutivo y un algoritmo de búsqueda local. Este método se adapta para resolver distintas variantes del *job shop scheduling problem*, tratando de minimizar tanto el tiempo total de ejecución como el retraso con respecto a fechas de entrega, con la posibilidad en ambos casos de que haya incertidumbre en la duración de las tareas. Los algoritmos diseñados se evalúan sobre un conocido conjunto de instancias, obteniendo resultados altamente competitivos.

Palabras clave: Inteligencia Artificial, metaheurísticas, problemas planificación tareas, incertidumbre

Abstract

Scheduling is a process that we humans perform daily, unconsciously, to carry out different jobs. However, in the industrial field, task scheduling cannot be carried out without prior analysis, since the decisions will affect the results and these will affect the economic gains.

The job shop scheduling problem is a classic problem that tries to model in a generic way those situations in which it is necessary to plan the execution of a series of operations on a finite set of resources.

Unfortunately, the complexity of the problem is NP-hard, which makes the use of techniques that ensure an optimal solution inviable. It is in this situation where metaheuristic techniques become relevant, because although they do not give any guarantee about the optimality of the returned solution, they allow to find sufficiently good solutions in a reasonable time. In addition, metaheuristic algorithms are especially interesting because their functioning is independent of the problem they are trying to solve, that is, they can be easily adapted to solve a wide variety of problems. For this reason, they are one of the most active research areas in the field of optimization problems.

In this work a memetic algorithm is proposed, a concrete type of metaheuristic that exploits the synergies between two different metaheuristic strategies, an evolutionary algorithm and a local search algorithm. This method is adapted to solve different variants of the job shop scheduling problem, trying to minimize the total execution time as well as the delay with regard to delivery dates, with the possibility in both cases of uncertainty in the duration of the tasks. The designed algorithms are evaluated on a well-known set of instances, obtaining highly competitive results.

Keywords: Artificial Intelligence, metaheuristics, scheduling, uncertainty

Índice

Índice de figuras **vii**

Índice de Tablas **viii**

1 Introducción **1**

1.1 Motivación . . . 1
1.2 Objetivos . . . 2
1.3 Organización . . . 3

2 Job Shop Problem **4**

2.1 Definición del problema . . . 4
2.2 Tipos de planificaciones . . . 5
2.3 Representación del problema . . . 6
2.3.1 Grafo disyuntivo para el makespan . . . 6
2.3.2 Grafo disyuntivo para el TWT . . . 9
2.3.3 Decodificación grafo disyuntivo . . . 9
2.4 Incertidumbre sobre los tiempos . . . 12
2.4.1 Triangular fuzzy numbers . . . 12
2.4.1.1 Operaciones aritméticas . . . 13
2.4.1.2 Operaciones relacionales . . . 13
2.4.2 Cambios sobre el grafo disyuntivo . . . 14

3 Metaheurísticas **15**

3.1 Búsqueda monoestado . . . 15
3.1.1 Funciones de vecindad . . . 15
3.1.1.1 Características generales de las vecindades . . . 16
3.1.1.2 Bloques críticos . . . 16
3.1.1.3 Vecindades consideradas . . . 17
3.1.1.3.1 CT . . . 18
3.1.1.3.2 CET . . . 19
3.1.1.3.3 CEI . . . 20
3.1.2 Búsqueda tabú . . . 22
3.1.2.1 Lista tabú . . . 23
3.1.2.2 Otros elementos de la búsqueda tabú . . . 24
3.1.2.2.1 Criterio de aspiración . . . 24
3.1.2.2.2 Detección de ciclos . . . 24
3.1.2.2.3 Todos los movimientos tabú . . . 24

3.2 Búsqueda multiestado 25
3.2.1 Algoritmos evolutivos 25
3.2.1.1 Codificación 26
3.2.1.2 Operadores de decodificación/evaluación 27
3.2.1.3 Operadores de selección 28
3.2.1.3.1 Ruleta 28
3.2.1.3.2 Parejas 29
3.2.1.4 Operadores de cruce 29
3.2.1.4.1 GOX 29
3.2.1.5 Operadores de mutación 30
3.2.1.5.1 Intercambio 31
3.2.1.6 Operadores de reemplazo 31
3.2.1.6.1 Generacional 31
3.2.1.6.2 Torneo 31
3.2.1.6.3 Elitismo 31
3.3 Algoritmos híbridos 32
3.3.1 Algoritmos meméticos 32

4 Desarrollo software **33**
4.1 Requisitos 33
4.1.1 Requisitos funcionales 33
4.1.2 Requisitos no funcionales 34
4.2 Metodología 34
4.3 Diseño 35
4.4 Pruebas 36
4.5 Documentación 37

5 Resultados experimentales **38**
5.1 Estudio experimental del JSP minimizando el TWT con tiempos de procesamiento deterministas 39
5.2 Estudio experimental del JSP minimizando el makespan con tiempos de procesamiento difusos 40
5.3 Estudio experimental del JSP minimizando el TWT con tiempos de procesamiento difusos 41

6 Conclusiones y trabajo futuro **44**
6.1 Conclusiones 44
6.2 Trabajo futuro 45
6.2.1 Reducción del tiempo de ejecución 45
6.2.2 Mejora de la calidad de las soluciones 46
6.3 Competencias del grado 46

Bibliografía **48**

Apéndices 51

A Estudio de sinergia 51

Índice de figuras

2.1 Tipos de planificaciones. 6
2.2 Grafo problema makespan. 7
2.3 Grafo solución makespan. 8
2.4 Grafo solución simplificado makespan. 8
2.5 Grafo problema TWT. 9
2.6 Grafo solución simplificado TWT. 11

3.1 Soluciones contraejemplo conectividad CET. 20
3.2 Codificación de un orden de tareas como una permutación con repetición. 27
3.3 Cromosomas para mostrar el comportamiento del operador de cruce GOX. 30
3.4 Operador cruce GOX en el caso de que el implante recaiga dentro. 30
3.5 Operador cruce GOX en el caso de que el implante se extienda por los bordes. . . . 31

Índice de Tablas

3.1 Definición problema contraejemplo conectividad CET. 20

5.2 Resultados del JSP minimizando el makespan con tiempos de procesamiento difusos. 41
5.1 Resultados del JSP minimizando el TWT con tiempos de procesamiento deterministas. 43
5.3 Resultados del JSP minimizando el TWT con tiempos de procesamiento difusos. . . 43

A.1 Resultados del estudio de sinergia. 54

Índice de algoritmos

2.1 Ordenamiento topológico. 10
2.2 Calculo EST. 11

3.1 Esquema de los algoritmos de búsqueda monoestado. 16
3.2 Esquema de la búsqueda tabú. 22
3.3 Esquema de los algoritmos de búsqueda multiestado. 25
3.4 Esquema de los algoritmos evolutivos. 26
3.5 Algoritmo G&T. 28
3.6 Esquema de los algoritmos meméticos. 32

Abreviaturas

BKS	best known solution
CEI	critical end insert
CET	critical end transpose
CT	critical transpose
DAG	directed acyclic graph
EST	earliest starting time
FIFO	first in first out
GOX	generalized order crossover
JSP	job shop scheduling problem
JSPTWT	job shop scheduling problem minimizing total weighted tardiness
OX	order crossover
RWS	roulette wheel selection
TFN	triangular fuzzy numbers
TWT	total weighted tardiness
UML	Unified Modeling Language

Capítulo 1

Introducción

En este capítulo se detallarán las motivaciones para la realización del trabajo (Sección 1.1), se enumerarán los objetivos que se pretenden conseguir (Sección 1.1) y se explicará la estructura seguida para organizar el resto del trabajo (Sección 1.3).

1.1 Motivación

El *job shop problem* es un problema de optimización combinatoria de gran interés para los investigadores, no solo por su alta complejidad computacional sino también por la gran cantidad de situaciones de la vida diaria que es capaz de modelar. De forma sencilla, el problema consiste en asignar una serie de operaciones a un conjunto finito de recursos de tal forma que se cumplan ciertas restricciones de precedencia, unas operaciones deben ejecutarse antes que otras, y de capacidad, los recursos solo pueden tener asignada una tarea al mismo tiempo.

Aunque el problema ha recibido mucha atención a lo largo de la historia, esta se ha centrado principalmente en reducir una única función objetivo, el tiempo total de procesamiento, conocido como *makespan*. En el pasado la industria también estaba de acuerdo en que esta era la métrica más importante, para ser capaces de terminar lo antes posible el producto y aumentar la producción; sin embargo en los últimos años se ha visto un cambio en la tendencía.

Los expertos en organización industrial han definido nuevas métricas que buscan, entre otras cosas, reducir el tiempo que los productos permanecen en los almacenes o reducir el consumo de energía de las máquinas. En este trabajo, aparte de la medida clásica del *makespan*, se considerará una conocida como *total weighted tardiness*, que mide el retraso con respecto a fechas de entrega, sabiendo que hay trabajos más prioritarios que otros. Esta medida resulta de gran importancia, pues el retraso de un trabajo en una cadena de producción puede tener un efecto cascada que retrase otros trabajos y al final se produzca una rotura de *stock* generando grandes pérdidas económicas para la empresa involucrada. Además, la medida es especialmente precisa porque tiene en consideración prioridades entre los trabajos, permitiendo retrasar un trabajo poco importante en beneficio de uno de importancia crítica.

Recientemente, el problema ha sido extendido para considerar incertidumbre en la duración de las operaciones. Aunque realizar un modelado con tiempos totalmente deterministas

simplifica el problema, en la vida real es muy difícil encontrar situaciones en las que los tiempos sean exactos y perfectamente conocidos de antemano. Lo más habitual es disponer de un posible rango de valores entre los que puede variar, habiendo además un valor más probable. Es importante que las técnicas de resolución empleadas tengan en cuenta esta información, si se dispone de ella, para de esta forma generar soluciones menos sensibles a posibles imprevistos.

Aunque el problema inicialmente se planteó para la organización de entornos de producción tipo taller, y de ahí proviene su nombre, su utilidad se extiende mucho más allá y abarca ámbitos tan distintos como la asignación de aviones a pistas de aterrizaje y despegue, la asignación de tareas a procesadores dentro de un sistema de computo o la fabricación de circuitos electrónicos [32]. Todas estas actividades no solo tienen en común unos tiempos que deben cumplirse con totales garantías, sino que es necesario poder actualizar la planificación en tiempo real ante posibles eventualidades.

Por esta última razón es necesario desarrollar algoritmos que no solo sean capaces de encontrar una solución, sino que lo hagan en un intervalo de tiempo razonable, de tal forma que puedan ser integrados incluso en sistemas que requieran de una respuesta inmediata. No obstante las técnicas de resolución que garantizan una solución óptima, debido a la complejidad del problema, no permiten dar respuesta a esta necesidad incluso en las instancias de menor tamaño. Es aquí donde cobran importancia las metaheurísticas, una familia de algoritmos que aunque no garantizan la óptimalidad de la solución retornada, permiten encontrar soluciones suficientemente buenas aún con un tiempo de ejecución muy reducido.

1.2 Objetivos

Los objetivos que se pretenden cubrir con este trabajo son:

1. Estudiar la bibliografía existente acerca del JSP para entender la complejidad del problema, el espacio de soluciones, las formas clásicas de modelar el problema y los algoritmos existentes para generación de planificaciones, de cara a explotar el conocimiento heurístico del problema en la búsqueda de soluciones.

2. Diseñar e implementar un algoritmo de búsqueda monoestado, que utilizando estructuras de vecindad, sea capaz de dar solución al problema.

3. Diseñar e implementar un algoritmo evolutivo, que sea capaz de encontrar una solución restringiéndose a un subconjunto del espacio de soluciones en el que esté garantizado que haya una solución óptima, para así aligerar la búsqueda.

4. Hibridar el algoritmo evolutivo con el algoritmo de búsqueda monoestado para dar como resultado un algoritmo memético que, mediante un equilibrio entre diversificación e intensificación, permita obtener mejores soluciones que ambos esquemas de búsqueda por separado.

5. Extender el modelo para considerar incertidumbre en la duración de las operaciones, haciendo uso de números difusos triangulares.

6. Evaluar los algoritmos propuestos, obteniendo resultados experimentales sobre un conjunto de instancias bien conocido. El objetivo final es que las técnicas propuestas sean competitivas con las mejores existentes hasta el momento.

1.3 Organización

A continuación se describe como se ha organizado el resto del trabajo.

En el Capítulo 2 se introduce el problema, las funciones objetivo que se van a optimizar y la forma de modelarlo. De esta manera queda totalmente definido el problema que se va a tratar. En este mismo capítulo se introducen los números difusos, el modelo escogido para considerar incertidumbre en la duración de las tareas.

En el Capítulo 3 se explican los algoritmos metaheurísticos considerados. Es decir, se explicarán las técnicas utilizadas para, explotando el conocimiento del problema disponible, encontrar una solución que satisfaga todas las restricciones impuestas.

En el Capítulo 4 se detallarán las decisiones tomadas a la hora de diseñar e implementar el programa. Este capítulo supone un resumen de las etapas más importantes del proceso de desarrollo software.

En el Capítulo 5 se mostrarán los resultados obtenidos aplicando las técnicas consideradas sobre un conjunto de instancias bien conocido.

Finalmente, en el Capítulo 6 se recogerán las conclusiones obtenidas del trabajo realizado.

Capítulo 2

Job Shop Problem

Durante esta sección se detallarán todos los aspectos del problema a tratar. Será uno de los puntos centrales de este trabajo pues es importante definir con claridad las características de lo que se quiere resolver antes de plantear posibles formas de solucionarlo. La Sección 2.1 define la estructura del problema; la Sección 2.2 detalla las propiedades del espacio de soluciones y la Sección 2.3 introduce la representación más habitual para el problema, conocida como grafo disyuntivo. Finalmente la Sección 2.4 extiende los conceptos presentados con anterioridad para considerar incertidumbre en la duración de las operaciones.

2.1 Definición del problema

El problema denominado *job shop problem* (JSP) consiste en planificar un conjunto de trabajos $J = \{J_1, J_2, \ldots, J_n\}$ en un conjunto de máquinas $M = \{M_1, M_2, \ldots, M_m\}$ de acuerdo a una serie de restricciones. Cada trabajo se compone de un conjunto de operaciones (también llamadas tareas) o_{jk} que deben ser ejecutadas de forma secuencial, donde el subíndice j indica el trabajo al que pertenecen, $1 \le j \le n$, y el subíndice k indica la posición que ocupan de forma relativa dentro de ese trabajo, $1 \le k \le m_j$ con $m_j \le m$. Dicho de otra forma las tareas están sujetas a restricciones de precedencia con el resto de operaciones de su mismo trabajo. No son las únicas restricciones que han de cumplirse, también existen restricciones de capacidad por las que una tarea o_{jk} debe mantener acceso exclusivo e interrumpido a la máquina $\mu_{jk} \in M$ en la que debe ejecutarse durante su tiempo de procesamiento $p_{jk} > 0$, con $\mu_{jk} \neq \mu_{jl}$ para $k \neq l$. El conjunto de todas las tareas se denota como O. Cada trabajo puede tener asociada una fecha límite $d_j > 0$ antes de la cual sería deseable que el trabajo estuviese terminado. Además no todos los trabajos son igual de importantes sino que tienen asociado un peso $w_j > 0$ que denota cuan prioritarios son.

Una solución al problema (también llamada planificación) es una asignación de tiempos de inicio a cada operación. Una solución es factible si dicha asignación cumple todas las restricciones del problema. Más formalmente, sean s_{jk} y $c_{jk} = s_{jk} + p_{jk}$ los tiempos de inicio y fin de cada tarea o_{jk}, entonces las restricciones de precedencia se cumplen si $s_{jk} + p_{jk} \le s_{j(k+1)}$ para $1 \le j \le n, 1 \le k < m_j$ y las restricciones de máquina se cumplen si $s_{jk} + p_{jk} \le s_{il} \vee s_{il} + p_{il} \le s_{jk}$ si $\mu_{jk} = \mu_{il}$, $1 \le i \le n, 1 \le j \le n, 1 \le k \le m_j, 1 \le l \le m_i$. Es habitual llamar solución al orden parcial

de ejecución de las tareas en cada máquina, ya que partir de este orden pueden establecerse los tiempos de inicio utilizando un planificador semiactivo, como se explica más adelante.

Falta determinar la calidad de una solución que dependerá de cuál sea el objetivo que se quiera obtener. El objetivo más habitual en la literatura es minimizar el *makespan* C_{max}, es decir, el tiempo total de procesamiento desde que comienza la primera tarea hasta que termina la última, que viene dado por $C_{max} = \text{máx}_{1 \leq i \leq n} C_i$ donde $C_i = c_{im_i}$ denota el tiempo de fin del trabajo J_i, $1 \leq i \leq n$. Otro objetivo de interés, que intenta medir el retraso con respecto a las fechas límite de los trabajos teniendo en cuenta que unos trabajos pueden ser más prioritarios que otros, es el llamado *total weighted tardiness* (TWT) definido como $TWT = \sum_{i=1}^{n} w_i T_i$ donde $T_i = \text{máx}\{0, C_i - d_i\}$ es el retraso o *tardiness* del trabajo J_i respecto a su fecha límite. Cabe señalar el hecho de que la medida TWT no penaliza aquellos trabajos que se terminan antes que su fecha límite.

Por comodidad, durante este trabajo, si no se menciona de forma explícita la función objetivo que se está utilizando, se empleará la notación JSP cuando se esté considerando el problema clásico, es decir, minimizar el *makespan* y JSPTWT cuando el objetivo sea minimizar el TWT.

Por último comentar la complejidad del problema. Sea n el número de trabajos y m el número de máquinas, entonces la minimización del *makespan* es un problema *NP-Hard* cuando $m \geq 2$ [13]. La minimación del TWT también es un problema *NP-Hard* bajo estas mismas condiciones, sin embargo es un problema más complejo pues en el caso de $m = 1$ ya es un problema *NP-Completo* [25]. Esta complejidad hace inviable el uso de métodos exactos para la resolución de la mayoría de instancias del problema salvo aquellas de menor tamaño y justifica el uso de técnicas metaheurísticas tal y como se va a hacer en este trabajo.

2.2 Tipos de planificaciones

Las planificaciones se clasifican en tres tipos diferentes: semiactivas, activas y *non-delay* [32].

Definición 2.2.1. *Planificación semiactiva*. Una planificación se considera semiactiva si y solo si no existe ninguna operación que pueda comenzar antes sin necesidad de cambiar el orden de las tareas de la máquina.

Definición 2.2.2. *Planificación activa*. Una planificación se llama activa si ninguna operación puede comenzar antes (cambiando el orden de ejecución dentro de las máquinas si es necesario) sin retrasar otra operación.

Definición 2.2.3. *Planificación non-delay*. Una planificación es *non-delay* si y solo si ninguna máquina está parada cuando existe una operación planificable en esa máquina.

Dadas estas definiciones, la clase de las planificaciones semiactivas incluye de forma estricta a las planificaciones activas y la clase de las planificaciones activas incluye de forma estricta las planificaciones *non-delay*. La Figura 2.1 muestra estas relaciones de inclusión.

Definición 2.2.4. *Medida regular*. Una medida o función objetivo se dice regular si es no decreciente en los tiempos de fin de los trabajos $C_1, \ldots, C_n$.

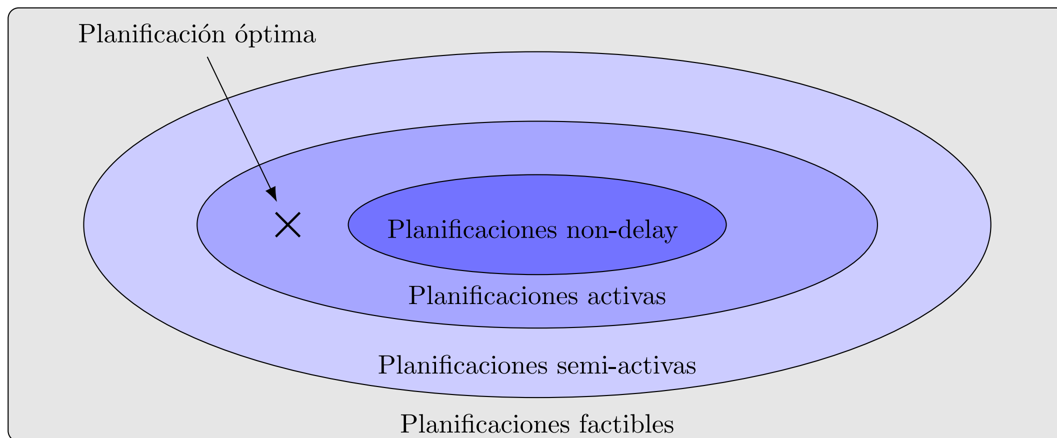

Figura 2.1: Tipos de planificaciones.

Notesé que dada la Definición 2.2.4 es trivial que tanto el *makespan* como el TWT son medidas regulares.

La relevancia de los espacios de planificaciones se debe a que puede probarse que algunos de ellos son dominantes para cualquier medida regular, es decir, contienen al menos una solución óptima, tal y como se enuncia en el Teorema 2.2.1. Este teorema, cuya demostración se puede encontrar en [12], permite limitar el tamaño del espacio de búsqueda, restringiéndola al subconjunto de soluciones semiactivas o el de activas, según convenga, ya que existe la garantía de poder encontrar al menos una solución óptima.

Teorema 2.2.1. *Dada una medida regular existe una solución óptima incluida en el espacio de planificaciones activas y una en el de semiactivas, que pueden o no coincidir y que pueden estar o no dentro del espacio de planificaciones non-delay.*

2.3 Representación del problema

Una vez definido el problema hay que encontrar una manera de representarlo. La manera más habitual de representar el JSP es la conocida como grafo disyuntivo [32]. A continuación se definirá esta representación para la función objetivo del *makespan* y se mostrará cómo puede extenderse al JSPTWT.

2.3.1 Grafo disyuntivo para el makespan

En el modelo de grafo disyuntivo cada nodo se corresponde con una operación. Las operaciones que pertenecen al mismo trabajo se conectan por aristas dirigidas, llamadas conjuntivas, representando las restricciones de precedencia. Las operaciones que tienen que ser procesadas en la misma máquina se conectan por aristas no dirigidas, llamadas disyuntivas, representando las restricciones de capacidad. Las aristas son no dirigidas porque inicialmente no se ha decidido ningún orden entre tareas en una misma máquina, es lo que busca la solución. Una arista no dirigida es equivalente a dos aristas dirigidas en sentidos contrarios y por ello se denominan

disyuntivas porque en una solución factible solo uno de los dos sentidos permanecerá como mucho.

Formalmente un grafo disyuntivo $G = (V, C, D)$ se compone de un conjunto de nodos o vértices V, un conjunto de aristas dirigidas (conjuntivas) C y un conjunto de aristas no dirigidas (disyuntivas) D.

El conjunto V aparte de contener todos los nodos representando a las operaciones del problema, también contiene dos nodos correspondientes a operaciones ficticias, el inicio s y el fin e, formalmente, $V = O \cup \{s, e\}$. El peso de cada nodo se corresponderá con el tiempo de procesamiento p_{jk} de la tarea o_{jk} que contiene, en el caso de los nodos de inicio y fin su peso es cero.

El conjunto C de las aristas conjuntivas se compone de todos los posibles pares ordenados de la forma (o_{jk}, o_{jk+1}), es decir, de pares donde el segundo componente es el sucesor en el trabajo del primer componente. Además contiene arcos que van del nodo inicio a la primera operación de cada trabajo, y de la última operación de cada trabajo al nodo fin, es decir, $C = \{(o_{jk}, o_{jk+1}) \mid 1 \leq j \leq n, 1 \leq k < m_j\} \cup \{(s, o_{j1}) \mid 1 \leq j \leq n\} \cup \{(o_{jm_j}, e) \mid 1 \leq j \leq n\}$.

El conjunto D de la aristas disyuntivas contiene todos los posibles pares de operaciones que se ejecutan en la misma máquina, como existen los dos sentidos, cada arista es un conjunto con dos pares ordenados, cada uno representando uno de los sentidos, $D = \{\{(o_{jk}, o_{il}), (o_{il}, o_{jk})\} \mid 1 \leq i \leq n, 1 \leq j \leq n, 1 \leq k \leq m_j, 1 \leq l \leq m_i, \mu_{jk} = \mu_{il}\}$.

Aunque se ha optado por incorporar los pesos en los nodos, se podría obtener una representación igualmente válida dando los pesos a las aristas. En este caso el peso de cada arista se correspondería con el peso el nodo de origen. Es habitual encontrar en la literatura esta representación alternativa. La Figura 2.2 muestra el grafo de un problema con tres trabajos y tres máquinas.

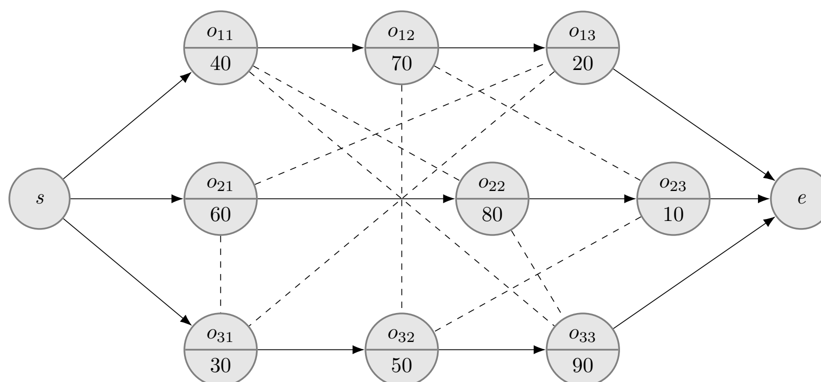

Figura 2.2: Grafo problema makespan. Las aristas conjuntivas se representan como una arista dirigida con la línea continua, mientras que las aristas disyuntivas se representan como una arista no dirigida con la línea discontinua. Se seguirá este mismo convenio durante todo el documento.

Para obtener una solución al problema a partir de esta representación, es necesario escoger un sentido para cada una de las aristas disyuntivas, dando un orden dentro de cada máquina. De esta manera se obtiene un grafo completamente dirigido (*digrafo*). En concreto, una solución

se dice factible si la selección es completa (para cada arista disyuntiva se selecciona un sentido) y el grafo resultante es un grafo acíclico dirigido, *directed acyclic graph* (DAG). La Figura 2.3 muestra el grafo solución del problema anteriormente presentado en la Figura 2.2.

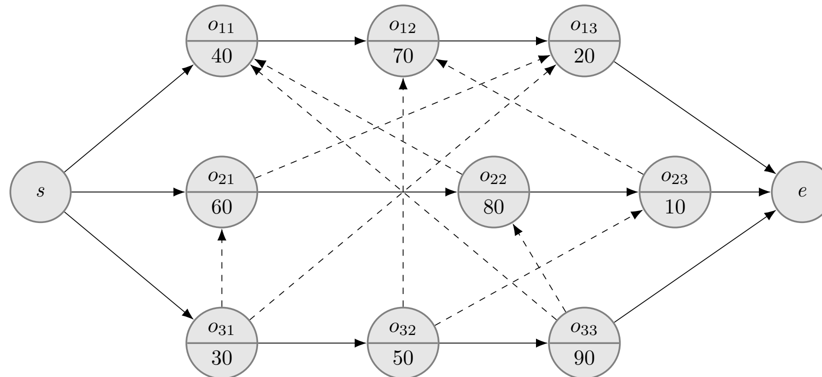

Figura 2.3: Grafo solución makespan.

Esta solución, aunque correcta, puede ser simplificada pues contiene aristas redundantes. La forma de detectar las aristas redundantes es aplicar la propiedad transitiva, si existe una forma de ir del nodo a al nodo b pasando por un nodo intermedio c y usando solo aristas disyuntivas, entonces la arista que va de a a b es redundante y puede ser eliminada. Nótese que en ningún caso la arista puede ir en sentido contrario de b a a porque entonces existiría un ciclo y no sería una solución factible. Aunque la simplificación no es necesaria desde el punto de vista de la correción de la solución, si que es necesaria para obtener la máxima eficiencia de los algoritmos. Durante el trabajo será a veces necesario referirse a uno u otro tipo de grafo, para diferenciarlos se denotará como G_σ al grafo solución simplificado de la solución σ y como $\overline{G}_\sigma$ a su grafo sin simplificar. Como en lo único que difieren ambos grafos es en el conjunto de aristas disyuntivas, se usará también la notación D_σ y $\overline{D}_\sigma$ de forma respectiva para el simplificado y el original. Nótese que la relación entre D_σ y $\overline{D}_\sigma$ es que el segundo es la clausura transitiva del primero. En la Figura 2.4 se puede ver el grafo simplificado de la solución de la Figura 2.3. A partir de este momento se supondrá por defecto la versión simplificada pues es la que se usará como entrada para los algoritmos.

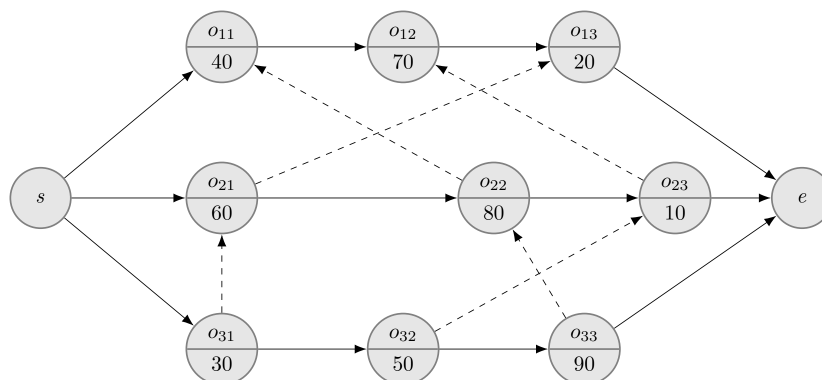

Figura 2.4: Grafo solución simplificado makespan.

2.3.2 Grafo disyuntivo para el TWT

Partiendo de la representación introducida para el *makespan*, si el objetivo a optimizar es el TWT, el modelo de grafo disyuntivo cambia ligeramente. El nodo final e se sustituye por dos nodos e_j y f_j por cada trabajo, de modo que ahora el conjunto de vértices o nodos es $V = O \cup \{e_j \mid 1 \leq j \leq n\} \cup \{f_j \mid 1 \leq j \leq n\}$. El nodo e_j tendrá como peso la fecha límite del trabajo cambiada de signo, $-d_j$, y f_j tendrá peso nulo. Sendos arcos dirigidos conectan la última tarea del trabajo o_{jm_j} con e_j y e_j con f_j. Además, se añaden arcos dirigidos desde el nodo inicial s a cada nodo final f_j, de modo que el conjunto de arcos conjuntivos es $C = \{(o_{jk}, o_{jk+1}) \mid 1 \leq j \leq n, 1 \leq k < m_j\} \cup \{(s, o_{j1}) \mid 1 \leq j \leq n\} \cup \{(o_{jm_j}, e_j) \mid 1 \leq j \leq n\} \cup \{(e_j, f_j) \mid 1 \leq j \leq n\} \cup \{(s, f_j) \mid 1 \leq j \leq n\}$, mientras que los arcos disyuntivos no cambian, es decir, $D = \{\{(o_{jk}, o_{il}), (o_{il}, o_{jk})\} \mid 1 \leq i \leq n, 1 \leq j \leq n, 1 \leq k \leq m_j, 1 \leq l \leq m_i, \mu_{jk} = \mu_{il}\}$. Los arcos desde el nodo inicial s a f_j aseguran que la longitud del camino más largo no pueda ser negativa tal y como requiere la definición de *tardiness*, esto tendrá relevancia a la hora de decodificar las soluciones. En la Figura 2.5 se puede ver la representación gráfica del mismo problema considerado anteriormente pero minimizando el TWT.

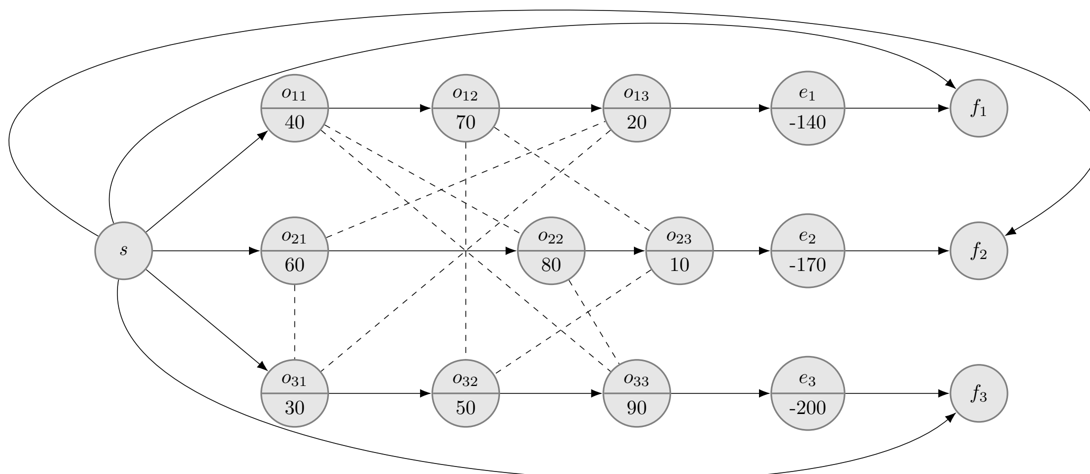

Figura 2.5: Grafo problema TWT.

Al igual que antes, encontrar una solución consiste es escoger un sentido para cada arista disyuntiva de tal forma que el grafo resultante sea un DAG. El grafo resultante también puede ser simplificado siguiendo el mismo criterio. El grafo solución ya simplificado se puede ver en la Figura 2.6.

2.3.3 Decodificación grafo disyuntivo

Al ser el grafo solución acíclico, impone un ordenamiento topológico en el grafo, permitiendo recorrer las operaciones en un orden en el que se respetan las restricciones. El Algoritmo 2.1 permite obtener el orden topológico del grafo en tiempo lineal.

A partir de este orden, es posible obtener de forma muy sencilla una solución semiactiva al problema. Para ello basta con asignar como tiempo de inicio s_{jk} de cada tarea o_{jk} su tiempo de inicio más pronto posible, *earliest starting time* (EST), est_{jk}.

Algoritmo 2.1: Ordenamiento topológico.

```
Data: O lista con todas las operaciones, P diccionario con el tiempo de procesamiento
      de cada operación, PJ diccionario con el predecesor de cada operación en el
      trabajo, PM diccionario con el predecesor de cada operación en la máquina, SJ
      diccionario con el sucesor de cada operación en el trabajo, SM diccionario con el
      sucesor de cada operación en la máquina
Result: T lista con las operaciones en orden topológico
Q cola ;                              ▷ Contiene las operaciones disponibles
A diccionario ;                         ▷ Contiene el número de predecesores
foreach t in O do
    if PJ[t] = null and PM[t] = null then
        Q.inserta(t) ; ▷ No tiene predecesores y se puede acceder directamente
    else if PJ[t] = null xor PM[t] = null then
        A[t] = 1;
    else
        A[t] = 2;
    end
end
while Q no vacio do
    t=Q.desapila();
    T.inserta(t);
    if SJ[t] ≠ null then
        A[SJ[t]]− = 1;
        if A[SJ[t]] = 0 then
            Q.insert(SJ[t]);
        end
    end
    if SM[t] ≠ null then
        A[SM[t]]− = 1;
        if A[SM[t]] = 0 then
            Q.insert(SM[t]);
        end
    end
end
if |O| ≠ |T| then
    return Error ;     ▷ La solución tiene ciclos y por tanto no es factible
end
return T
```

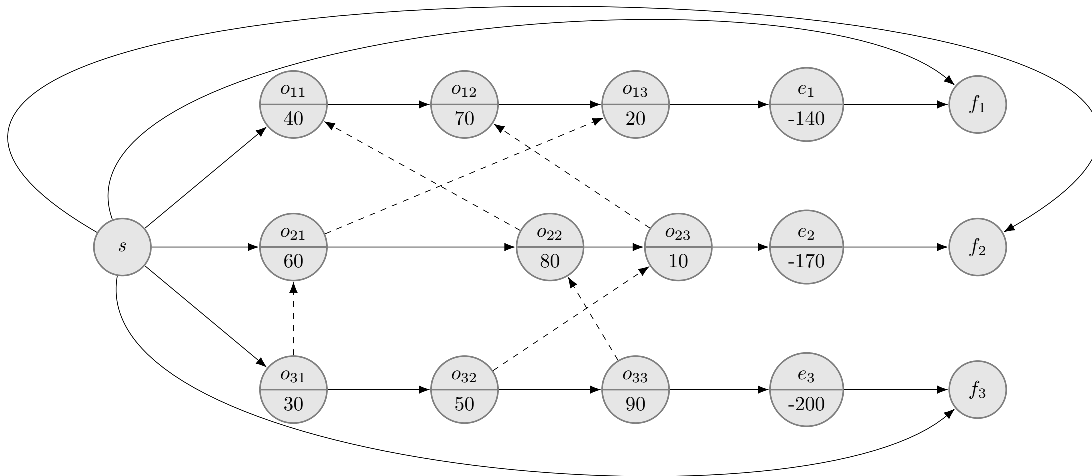

Figura 2.6: Grafo solución simplificado TWT.

Dada una solución σ o, equivalentemente, un grafo solución G_σ, es posible calcular el est_{jk} de una tarea o_{jk} sin más que propagar restricciones en el grafo a partir de la tarea inicial s. Dicho de otro modo, si $PM_{jk}(\sigma)$ denota el predecesor en la máquina de o_{jk} para la solución σ, entonces $est_{jk} = max\{est_{j(k-1)} + p_{j(k-1)}, est_{PM_{jk}(\sigma)} + p_{PM_{jk}(\sigma)}\}$ donde $est_{j(k-1)} + p_{j(k-1)} = 0$ si se trata de la primera operación en el trabajo ($k = 1$) y $est_{PM_{jk}(\sigma)} + p_{PM_{jk}(\sigma)} = 0$ si se trata de la primera operación en su máquina.

El Algoritmo 2.2 hace uso de esta definición para de forma iterativa calcular el EST de todas las tareas.

Algoritmo 2.2: Calculo EST.

Input: T lista con todas las operaciones en orden topológico, P diccionario con el tiempo de procesamiento de cada operación, PJ diccionario con el predecesor de cada operación en el trabajo, PM diccionario con el predecesor de cada operación en la máquina
Output: S diccionario con el EST de cada tarea

```
foreach t in T do
    S[t] = 0;
    if PJ[t] ≠ null then
        S[t] = max(S[t], S[PJ[t]] + P[PJ[t]]);
    end
    if PM[t] ≠ null then
        S[t] = max(S[t], S[PM[t]] + P[PM[t]]);
    end
end
return S
```

Conociendo el EST de las tareas el primer instante en que puede completarse cada trabajo es $ect_j = est_{jm_j} + p_{jm_j}$. Es decir, si se fija el tiempo de inicio de cada tarea como el primer instante en que puede comenzar, el tiempo de fin c_{jk} coincidirá con el primer instante en que puede completarse y, a partir de aquí, se podrán calcular tanto el *makespan* como el TWT según las expresiones definidas en la Sección 2.1.

Dada esta forma de decodificar las soluciones, la representación escogida cobra aún más relevancia, pues el valor del *makespan* es la longitud del camino más largo del nodo inicio s al nodo fin e, es decir, el camino crítico. Por otro lado, el *tardiness* T_j de cada trabajo es la longitud del camino más largo desde el nodo inicio s hasta su nodo fin f_j. Nótese que en ninguno de los dos casos el camino crítico es necesariamente único.

Las planificaciones obtenidas haciendo uso de este algoritmo son planificaciones semiactivas porque siempre se les da el tiempo de inicio lo más pronto posible preservando el orden impuesto por el grafo.

2.4 Incertidumbre sobre los tiempos

Durante las secciones anteriores se introdujo el JSP y su variante considerando fechas límite el JSPTWT. En esta sección se van a extender ambos problemas para considerar incertidumbre en la duración de las operaciones. La justificación detrás de esta extensión es que lo más habitual es que no se conozca con certeza cuanto tiempo va a ser necesario para realizar una tarea sino que se pueda dar una cota inferior, una superior y un valor más probable. Esto nos lleva a considerar un subconjunto de los números difusos, más conocidos por su nomenclatura en inglés *fuzzy numbers*, conocido como números difusos triangulares (TFN, del inglés *triangular fuzzy numbers*) [10]. En este trabajo se considerarán como TFN únicamente los tiempos de procesado de las tareas, las fechas límite seguirán siendo números deterministas.

En la Sección 2.4.1 se introducen los TFN y la forma de operar con ellos. Posteriormente, en la Sección 2.4.2, se procederá a realizar una serie de ajustes sobre el modelado del problema realizado anteriormente.

2.4.1 Triangular fuzzy numbers

En esta sección se explicará que es un TFN y cuál es la forma de operar con ellos, como es un concepto muy amplio se introducirán solo aquellos conceptos relevantes para su aplicación a los problemas de planificación [31].

Un número difuso permite referirse en lugar de a un valor concreto, a un conjunto de valores cada uno de los cuales tiene asociado un grado de pertenencia. A continuación se da la definición matemática formal.

Definición 2.4.1. *Cantidad difusa*. Una *fuzzy quantity* o cantidad difusa Q es un conjunto difuso sobre los reales con función de pertenencia $\mu_Q : \mathbb{R} \to [0,1]$. Los $\alpha-cortes$ de una cantidad difusa vienen dados por $Q_\alpha = \{r \in \mathbb{R} \mid \mu_Q(r) \geq \alpha\}, \alpha \in (0,1]$ y su *soporte* se define como $Q_0 = \{r \in \mathbb{R} \mid \mu_Q(r) > 0\}$.

Definición 2.4.2. *Intervalo difuso*. Un *fuzzy interval* o intervalo difuso A es una *fuzzy quantity* cuyos α-cortes son intervalos (acotados o no).

Definición 2.4.3. *Número difuso*. Un *fuzzy number* o número difuso M es un *fuzzy interval* cuyo soporte es compacto (cerrado y acotado), su valor modal es único y sus α-cortes son cerrados (denotados como $M_\alpha = [\underline{m}_\alpha, \overline{m}_\alpha]$).

El modelo más sencillo, y el cual se usará en este trabajo, son los ya mencionados números difusos triangulares.

Definición 2.4.4. *Número difuso triangular*. Un TFN es un número difuso con un intervalo $[a^1, a^3]$ de posibles valores y un valor modal a^2. Un TFN A se representa como $A = (a^1, a^2, a^3)$ y su función de pertenencia viene dada según la expresión de la Ecuación 2.1.

$$\mu_A(x) = \begin{cases} \frac{x-a^1}{a^2-a^1} & a^1 \leq x \leq a^2 \\ \frac{x-a^3}{a^2-a^3} & a^2 \leq x \leq a^3 \\ 0 & x < a^1 \vee a^3 < x \end{cases} \tag{2.1}$$

2.4.1.1 Operaciones aritméticas

Para el JSP minimizando el *makespan* hacen falta únicamente la suma y el máximo y minimizando el TWT hace falta además la resta con un número determinista (porque las fechas límite consideradas son valores reales). Todas ellas se obtienen a partir de las operaciones sobre los números reales aplicando el principio de extensión [37]. De acuerdo a esto la suma de dos TFN M, N viene dada por la Ecuación 2.2.

$$M + N = (m^1 + n^1, m^2 + n^2, m^3 + n^3) \tag{2.2}$$

La resta de un TFN M con un número determinista a se define en la Ecuación 2.3.

$$M - a = (m^1 - a, m^2 - a, m^3 - a) \tag{2.3}$$

El máximo entre dos TFN es computacionalmente muy complejo y trabajar con él es poco práctico ya que el resultado aunque está garantizado que sea un número difuso puede que no sea un TFN, dicho de otra manera, el conjunto de los TFN no es cerrado bajo esta operación. Por ello, y siguiendo la literatura [11], se va a aproximar el máximo entre dos TFN como el resultado de evaluar el máximo en cada una de las componentes por separado, tal y como se define en la Ecuación 2.4. Se ha optado por esta aproximación frente a otras [24] porque mantiene mejores propiedades desde el punto de vista teórico, concretamente mantiene el soporte y el valor modal del máximo no aproximado [31].

$$\text{máx}(M, N) \approx (\text{máx}(m^1, n^1), \text{máx}(m^2, n^2), \text{máx}(m^3, n^3)) \tag{2.4}$$

2.4.1.2 Operaciones relacionales

No existe una relación de orden total para los TFN pero es una operación necesaria para su uso en múltiples aplicaciones. Por ello en la literatura se han propuesto diferentes alternativas [5, 6]. Para este trabajo se van a comparar los TFN en función de su valor esperado. La función de pertenencia de un *fuzzy number* se puede ver como una distribución de posibilidad sobre los reales por lo que su valor esperado se puede definir según la Ecuación 2.5 [18].

$$E[M] = \frac{m^1 + 2m^2 + m^3}{4} \tag{2.5}$$

Así se induce una relación de orden total $\leq$ en el conjunto de los TFN de tal forma que $M \leq N$ si y solo si $E[M] \leq E[N]$.

2.4.2 Cambios sobre el grafo disyuntivo

La representación de grafo disyuntivo se puede extender para usar TFN sin más que cambiar los pesos de los nodos, que ahora pasan a ser TFN, salvo en el caso de los nodos e_i para el TWT (que mantienen su peso $-d_i$). Como la estructura del grafo es la misma, el recorrido en orden topológico tampoco cambia y, gracias a la aritmética presentada en la Sección 2.4.1.1, la manera en que se propagan las restricciones en el grafo solución sigue siendo válida. En concreto, sigue siendo posible calcular el EST haciendo uso del Algoritmo 2.2, con la única salvedad de que ahora los tiempos de inicio s_{jk} y de fin c_{jk} de cada tarea o_{jk}, y por tanto los valores del *makespan* y TWT, serán TFN.

Cabe notar que esta extensión es totalmente coherente con el caso determinista, lo que permite trabajar con problemas en los que hubiese incertidumbre en algunas tareas y en otras no e incluso, en el caso extremo de que todos los TFN sean en realidad números deterministas, es decir, sus tres componentes tengan el mismo valor. Sin embargo, en este caso, el valor del *makespan* no coincide necesariamente con el tiempo de fin de un trabajo, sino que representa el menor tiempo de fin posible del problema, el más plausible y el mayor tiempo de fin posible; igualmente la *tardiness* de cada trabajo representa la menor *tardiness* posible para dicho trabajo, la más plausible y la mayor posible. Además, el resultado que se obtiene de realizar esta extensión no tiene por que coincidir con el que se obtendría si se aplicase el caso determinista haciendo uso de los valores esperados de cada tarea.

Más complicado es el concepto de camino crítico. Para ello, siguiendo la propuesta de [17], se descompondrá el grafo en tres niveles, uno por cada componente de los TFN, de tal manera que un camino es crítico si es crítico en alguno de los niveles, que son grafos deterministas en los que este concepto está definido. Con esta definición, cada componente del *makespan* calculado mediante propagación de restricciones coincide con la longitud de un camino crítico en el nivel correspondiente y cada componente de la *tardiness* T_i de cada trabajo J_i coincide con la longitud de un camino crítico del inicio a f_i en el nivel correspondiente. Cabe señalar el hecho de que no se trabaja con tres grafos distintos, sino con tres niveles que forman parte del mismo grafo, es decir son tres niveles isomorfos y el cambio de una arista en el grafo afecta a todos los niveles.

Las definiciones de tipos de planificación presentadas en la Sección 2.2 permanecen también tal y como están, aplicándose componente a componente [30].

Tal y como se ha definido, el valor que se obtiene tanto para el *makespan* como para el TWT es un TFN, sin embargo, para decidir si una solución es mejor que otra necesariamente hay que realizar una comparación, que tal y como se definió en la Sección 2.4.1.2 hace uso de su valor esperado.

Capítulo 3

Metaheurísticas

En este capítulo se explicarán las técnicas consideradas para dar solución al problema. La Sección 3.1 se centrará en los algoritmos de búsqueda monoestado, que consideran una única solución al mismo tiempo y la Sección 3.2, detallará el funcionamiento de los algoritmos de búsqueda multiestado, que consideran muchas soluciones al mismo tiempo.

3.1 Búsqueda monoestado

El funcionamiento de los algoritmos de búsqueda monoestado es sencillo, se parte de una solución inicial cualquiera y mediante una función de vecindad se obtiene un conjunto de soluciones candidatas a las que moverse [36]. En función de un criterio preestablecido el algoritmo escoge una de ellas, que pasa a ser la actual, y repite el proceso. Esta secuencia se repite de forma continuada hasta que se cumple un criterio de parada. Para decidir entre las distintas soluciones el algoritmo puede evaluarlas, es decir, obtener su calidad. El proceso de evaluación puede tener distintos grados de exactitud, pudiéndose optar por usar estimaciones si este es muy costoso. Además el algoritmo también puede tener en cuenta las características de las soluciones, el conjunto de atributos que la hacen ser única y diferenciarse del resto. Este esquema general se muestra en el Algoritmo 3.1, en función del comportamiento de cada una de las partes se obtiene un algoritmo de búsqueda monoestado u otro.

En la Sección 3.1.1 se profundizará en el concepto de vecindad y se definirán las que se han considerado a la hora de resolver el JSP. Posteriormente, en la Sección 3.1.2 se describirá la búsqueda tabú, el algoritmo monoestado utilizado.

3.1.1 Funciones de vecindad

Como se ha dicho, una estructura de vecindad N permite, a partir de una solución σ, obtener un conjunto de soluciones vecinas o adyacentes. Durante los siguientes apartados se indicarán cuales son las características que hacen buena una función de vecindad (Sección 3.1.1.1), se introducirá el concepto de bloques críticos (Sección 3.1.1.2), un elemento indispensable del JSP y por último se definirán las funciones de vecindad usadas en el trabajo (Sección 3.1.1.3).

Algoritmo 3.1: Esquema de los algoritmos de búsqueda monoestado.

```
Data: P problema a resolver
Result: M mejor solución encontrada
S = ObténSoluciónInicial(P);
M = S;
while not CriterioParada() do
    N = ObténVecinos(S);
    S = SeleccionaVecino(N);
    if S > M then
        M=S;
    end
end
return M
```

3.1.1.1 Características generales de las vecindades

A la hora de definir estructuras de vecindad, son varias las características a considerar [27]:

- **Correlación:** los vecinos tienen que ser semejantes a la solución original, es decir, deben diferir en unos pocos atributos. Si las soluciones son muy distintas, el algoritmo de búsqueda que haga uso de la función dará "saltos" en el espacio de soluciones y no intensificará la búsqueda adecuadamente.

- **Factibilidad:** lo preferible es que los vecinos sean ya soluciones factibles, en caso contrario será necesario comprobar su factibilidad, lo cual supone un consumo adicional de recursos computacionales. Si las soluciones no son factibles existen dos opciones, descartarlas o repararlas.

- **Conectividad:** la propiedad de conectividad determina si es posible partiendo de una solución cualquiera llegar a una solución óptima en un número finito de pasos aplicando de forma repetida la función de vecindad.

- **Tamaño:** debe ser suficientemente grande como para dar cabida a soluciones que puedan mejorar la solución actual, pero también suficientemente pequeño para no gastar demasiados recursos en cada etapa de búsqueda.

- **Mejora:** la probabilidad de que las soluciones generadas mejoren la actual debe ser alta.

3.1.1.2 Bloques críticos

Recordando la representación de grafo disyuntivo presentada anteriormente, se concluyó que, en el caso de minimizar el *makespan*, el valor de este era la longitud del camino crítico del nodo inicio al nodo final y en el caso de minimizar el TWT, la *tardiness* de cada trabajo J_i venía determinada por la longitud del camino crítico del nodo inicio a cada nodo final f_i. En el caso de que los tiempos de procesado de las tareas fuesen TFN, se utilizaba un grafo con tres niveles diferentes, sobre cada uno de los cuales se realizaban los cálculos de forma independiente, para

después componer un único resultado. Tambíen se ha mencionado que los caminos críticos no tienen por que ser únicos.

Los arcos y tareas que forman parte de un camino crítico reciben de forma respectiva el nombre de arcos críticos y tareas críticas. Un bloque crítico se define como la mayor secuencia posible de aristas disyuntivas críticas adyacentes. Dicho de otra manera, el conjunto de todos los bloques críticos son los subgrafos que resultan de eliminar todas las aristas conjuntivas y aquellas aristas disyuntivas que no son críticas.

En el caso del JSPTWT las tareas, aristas y bloques críticos están asociados a un trabajo, pudiendo estar asociado a varios si sus caminos críticos se superponen. Además, en el caso de considerar problemas que hagan uso de los TFN, también están vínculados a uno de los níveles del grafo. No obstante, lo más habitual será referirse a todos los bloques críticos de una solución σ como un conjunto B_σ sin hacer distinciones.

Estos conceptos tienen gran importancia a la hora de definir las vecindades, pues permiten restringir su tamaño. Para mejorar la calidad de una solución hay que reducir la longitud de su camino crítico y para ello hay que realizar cambios solo sobre las aristas que le afectan, pues de otra forma el camino crítico permanecería y en el peor caso se podría generar un camino más largo. Este hecho se enuncia en la Proposición 3.1.1.

Proposición 3.1.1. *Sean σ una solución y G_σ su grafo asociado y sea σ' una solución factible obtenida a partir de σ modificando únicamente arcos no críticos en G_σ. Entonces, σ' no puede mejorar a σ, es decir:*

$$f(\sigma) \leq f(\sigma')$$

para cualquier medida regular de rendimiento f (en especial, $f = C_{max}$ o $f = TWT$) y donde $\leq$, en el caso de valores difusos, se refiere al orden establecido según valores esperados en la Sección 2.4.1.2.

3.1.1.3 Vecindades consideradas

El resultado de la Proposición 3.1.1 sugiere definir estructuras de vecindad basadas en modificar arcos críticos disyuntivos. Esta es precisamente la motivación para las tres estructuras de vecindad consideradas en este trabajo, que se presentarán a continuación. Las dos primeras consisten en modificar el orden relativo de dos tareas críticas consecutivas en una máquina, es decir, invertir un arco crítico disyuntivo. La tercera produce un cambio mayor que aun así puede verse como una concatenación de inversiones de arcos críticos disyuntivos.

Antes de comenzar conviene matizar que, anteriormente, dada una solución σ, se habló de la existencia de un grafo solución asociado $\overline{G}_\sigma$ y una simplificación de este G_σ. La acción de invertir una arista solo se puede llevar a cabo de forma directa sobre el grafo sin simplificar $\overline{G}_\sigma$. Para invertir un arco crítico (u,v) en el grafo simplificado G_σ se elimina (u,v) y se inserta (v,u) pero también hay que eliminar las aristas $(PM_u(\sigma),u)$ y $(v,SM_v(\sigma))$ e insertar $(PM_u(\sigma),v)$ y $(u,SM_v(\sigma))$ para reflejar la inversión del orden relativo de ambas tareas en la máquina. En el caso de que $PM_u(\sigma)$ o $SM_v(\sigma)$ no existan, bastará con eliminar $(v,SM_v(\sigma))$ e insertar $(u,SM_v(\sigma))$ o eliminar $(PM_u(\sigma),u)$ e insertar $(PM_u(\sigma),v)$ de forma respectiva.

3.1.1.3.1 CT

La función de vecindad *critical transpose* (CT) fue introducida por van Laarhoven para el JSP con *makespan* como medida de rendimiento [23]. Consiste en invertir un único arco crítico, es decir, cambiar el orden de dos tareas consecutivas pertenecientes a la misma máquina. En la Definición 3.1.1 se establece formalmente el conjunto de soluciones a las que se puede mover aplicando esta vecindad.

Definición 3.1.1. *Vecindad CT*. Sean σ una solución, G_σ el grafo solución asociado y σ_a la solución que se obtiene invirtiendo un arco disyuntivo a en G_σ. Entonces la vecindad CT viene dada por:

$$CT(\sigma) = \{\sigma_a \mid a \text{ es un arco disyuntivo crítico en } G_\sigma\}$$

En cuanto a sus características, primero se demostrará su factibilidad. El Teorema 3.1.1 asegura que la inversión de un arco crítico siempre da lugar a una solución factible. La demostración original para el JSP de [23] se generaliza para incluir el caso en que se minimiza el TWT y el caso en que los tiempos de procesamiento son TFN.

Teorema 3.1.1. *Sean σ una solución factible y G_σ su grafo solución asociado. Entonces, la inversión de un arco disyuntivo crítico (u,v) siempre da lugar a una solución factible.*

Demostración. Por reducción al absurdo. Supóngase que la solución σ' resultante de la inversión del arco (u,v) no es factible, entonces, su grafo asociado $G_{\sigma'}$ debe tener un ciclo. Como σ es factible, el grafo G_σ es acíclico y por tanto la arista (v,u) tiene que formar parte del ciclo en $G_{\sigma'}$. La existencia del ciclo implica la existencia de un camino $(u,\ldots,v)$ en $G_{\sigma'}$ con más de una arista. Sin embargo, este camino también debe existir en G_σ y necesariamente es más largo que el que solo incluye (u,v), con lo que se contradice la hipótesis inicial de que (u,v) es crítico.

La demostración es correcta tanto si se consideran números deterministas como TFN porque como los niveles del grafo son isomorfos, si no existe un ciclo en uno de ellos no puede existir en ninguno. □

Corolario 3.1.1. *Dada una solución factible σ, su vecindad CT solo contiene soluciones factibles, es decir, $CT(\sigma) \subseteq \Sigma$ donde Σ denota el conjunto de soluciones factibles.*

La vecindad también tiene la propiedad de conectividad, que garantiza que una búsqueda local utilizando esta vecindad puede alcanzar un óptimo partiendo de cualquier solución. Para demostrarlo primero hace falta el Lema 3.1.1.

Lema 3.1.1. *Sea σ una solución cualquiera y σ_0 una solución óptima cualquiera. Si σ no es óptimo entonces el conjunto $K_\sigma(\sigma_0) = \{(u,v) \in D_\sigma \mid (u,v) \text{ es crítico } \wedge (v,u) \in \overline{D}_{\sigma_0}\} \neq \emptyset$*

Demostración. La demostración consta de dos partes. La primera asegura que D_σ siempre contiene arcos críticos a no ser que sea óptima. La segunda, que hay arcos críticos en D_σ que no están en $\overline{D}_{\sigma_0}$ a menos que σ sea óptima.

1. Si D_σ no tiene ningún arco crítico entonces quiere decir que el camino crítico pasa solo por aristas conjuntivas entre tareas de un mismo trabajo, que no pueden ser cambiadas y la solución es por tanto óptima.

2. Supóngase que todos los arcos críticos de D_σ están incluidos en $\overline{D}_{\sigma_0}$. Entonces cualquier camino crítico P en G_σ es un camino Q en $\overline{G}_{\sigma_0}$. Todo camino crítico R en $\overline{G}_{\sigma_0}$ también es un camino crítico en G_{σ_0}. Por lo tanto, si L_s denota la longitud del camino s, $L_R \leq L_P$ (por definición de solución óptima) y $L_R \geq L_Q = L_P$ (por definición de camino crítico) por lo que L_R tiene que ser igual a L_P lo que implica que σ es óptima.

Nótese que ambas partes son ciertas con independencia de si se usan números deterministas o TFN puesto que las relaciones utilizadas se cumplen para ambos. □

Ahora se puede demostrar la conectividad (Teorema 3.1.2).

Teorema 3.1.2. *Para cualquier solución no óptima σ es posible encontrar una secuencia de transiciones de CT que la lleven a una solución óptima.*

Demostración. Dada σ_0 una solución óptima cualquiera, se construye una secuencia de transiciones $(\lambda_0, \lambda_1, \ldots)$ de la siguiente manera:

- $\lambda_0 = \sigma$
- λ_{i+1} se construye a partir de λ_i invirtiendo una arista disyuntiva que también se encuentre en $K_{\lambda_i}(\sigma_0)$. Esto se puede hacer sin provocar ciclos gracias a la propiedad de factibilidad.

Al cabo de $k = |\overline{D}_{\sigma_0} \setminus \overline{D}_\sigma|$ pasos, es decir, el número de aristas que tienen en distinta orientación los grafos $\overline{D}_{\sigma_0}$ y $\overline{D}_{\sigma_0}$, todas las aristas pasarán a estar en la misma orientación y por tanto $K_{\lambda_K}(\sigma_0) = \emptyset$. De esta forma según el Lema 3.1.1 λ_k es una solución óptima. □

3.1.1.3.2 CET

La función de vecindad *critical end transpose* (CET) fue propuesta por Nowicki y Smutnicki [29] como una reducción de CT ya que invertir un arco crítico solo puede producir mejoras en la calidad de las soluciones cuando el arco intercambiado se encuentra en un extremo del bloque crítico. La razón es que cuando se invierte un arco que está dentro del bloque, el tiempo de inicio de la tarea siguiente no se ve afectado y por tanto no cambia la longitud del camino. La vecindad CET por tanto lo que hace es intercambiar solo uno de los dos extremos de un bloque crítico (en el caso de que el bloque crítico esté compuesto de una sola arista ambos extremos se consideran el mismo). El conjunto de soluciones a la que se puede mover aplicando esta vecindad se establece formalmente en la Definición 3.1.2

Definición 3.1.2. *Vecindad CET*. Sean σ una solución, G_σ el grafo solución asociado, σ_a la solución que se obtiene invirtiendo un arco disyuntivo a en G_σ, $B_\sigma = \{b_1, \ldots, b_r\}$ el conjunto de todos los bloques críticos de σ y $b_i = (u_{i1}, \ldots, u_{is_i})$ la secuencia de operaciones del bloque crítico b_i donde s_i representa el número de operaciones en el bloque. Entonces la vecindad CET viene dada por:

$$CET(\sigma) = \{\sigma_a \mid a \in \{(u_{i1}, u_{i2}), (u_{i(s_i-1)}, u_{is_i})\}, b_i \in B_\sigma\}$$

En cuanto a su factibilidad nótese que CET⊆CT por lo que dada la factibilidad de CT, también han de ser factibles todas las soluciones de CET.

Esta vecindad pese a ser incluida en CT no mantiene la propiedad de conectividad. Basta considerar el ejemplo de la Tabla 3.1. En la Figura 3.1 se muestra la instancia inicial y la óptima a la que no se puede acceder pues requiere colocar la tarea o_{14} en la última posición de la máquina 1, lo cual no es posible intercambiando solo los extremos.

Tabla 3.1: Definición problema contraejemplo conectividad CET.

Trabajo j	o_{j1}		o_{j2}	
	p_{j1}	$\mu_j 1$	$p_j 2$	$\mu_j 2$
1	4k	1	2k	2
2	4k	1	3k	2
3	4k	1	3k	2
4	4k	1	k	2

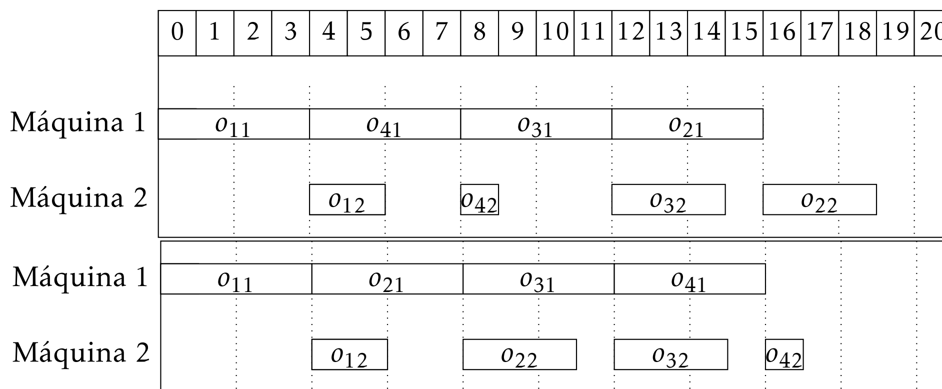

Figura 3.1: Diagrama de Gantt con la solución inicial (arriba) y óptima (abajo).

3.1.1.3.3 CEI

La función de vecindad *critical end insert* (CEI) fue introducida por Dell'Amico y Trubian [8]. Consiste en mover una tarea en el interior de un bloque crítico lo más adelante o detrás posible en el bloque de tal forma que la solución resultante siga siendo factible. Formalmente, el conjunto de soluciones a las que se puede mover aplicando la vecindad CEI se determina en la Definición 3.1.3.

Definición 3.1.3. *Vecindad CEI.* Sean σ una solución y B_σ el conjunto de todos los bloques críticos de σ. Dado un bloque $b \in B_\sigma$ de longitud $s > 1$, si $I(b,i,\underline{e}(i))$ (respectivamente, $I(b,i,\overline{e}(i))$) denota la inserción de la i-ésima tarea de b en la posición más cercana al inicio (respectivamente, final) que resulta factible, es decir:

$$\underline{e}(i) = argmin\{j < i \mid I(b,i,j) \text{ es factible }\}$$
$$\overline{e}(i) = argmax\{j > i \mid I(b,i,j) \text{ es factible }\}$$

Entonces, el conjunto de movimientos de CEI en b se define como:

$$M(b) = \{I(b,i,\underline{e}(i)) \mid 1 \leq i \leq s\} \cup \{I(b,i,\overline{e}(i)) \mid 1 \leq i \leq s\}$$

y la vecindad CEI viene dada por:

$$CEI(\sigma) = \{\sigma_m \mid m \in M(b), b \in B_\sigma\}$$

donde σ_m es el resultado de realizar en σ el movimiento m.

Debido a que la propia vecindad exige que el vecino que se forma sea factible, su factibilidad está asegurada. Además la vecindad tiene la propiedad de conectividad, tal y como se indica en el Teorema 3.1.3 enunciado en [8] para el JSP clásico y que aquí se generaliza al JSPTWT y al caso de duraciones difusas.

Teorema 3.1.3. *Para cada solución factible σ, es posible construir una secuencia finita de movimientos usando la vecindad CEI que lleve de σ a un óptimo global.*

Demostración. Sean σ la solución actual, $\overline{G}_\sigma$ su grafo asociado sin simplificar y P un camino crítico en $\overline{G}_\sigma$ que contiene q bloques críticos $\{b_1, b_2, \ldots, b_q\}$. Sea $b_i = (u_{i1}, \ldots, u_{is_i})$ la secuencia de operaciones del bloque crítico b_i donde s_i representa el número de operaciones, $1 \leq i \leq q$. Dada una solución óptima σ_0 y su grafo sin simplificar $\overline{G}_{\sigma_0}$, si las operaciones $\{b_i \setminus \{u_{i1}, u_{is_i}\}\}$ del interior del bloque están planificadas entre u_{i1} y u_{is_i} en σ_0 para todo bloque $b_i \in B_\sigma$, entonces la solución actual también es óptima porque la longitud de un camino crítico en σ_0 no puede ser menor que la longitud de P. Si la anterior condición no se cumple, entonces existe un bloque b_i y una operación $k \in b_i$ tales que el arco (u_{i1}, k) o el arco (k, u_{is_i}) de $\overline{G}_\sigma$ están invertidos en $\overline{G}_{\sigma_0}$. Sin pérdida de generalidad se considerará únicamente que el arco (u_{i1}, k) está invertido puesto que la argumentación para el otro caso es análoga. Se quiere probar que en la vecindad CEI hay un movimiento que reduce el número de arcos con diferente orientación (distancia) entre $\overline{G}_\sigma$ y $\overline{G}_{\sigma_0}$. Sea $k \in b_i$ la primera operación tal que el arco (u_{i1}, k) está invertido en $\overline{G}_{\sigma_0}$. Si la solución obtenida como resultado insertar k la primera en el bloque es factible, se puede conseguir con un movimiento de CEI, lo que provoca que la distancia entre $\overline{G}_\sigma$ y $\overline{G}_{\sigma_0}$ se reduzca. En otro caso, sea $j \in b_i$ la última operación que precede a k en el bloque b_i tal que la inversión del arco (j,k) y todos los arcos (h,k) para cada $h \in b_i$ planificado entre j y k produce una solución no factible. Sea b' el conjunto de operaciones del bloque b_i entre j y k. Nótese que b' contiene al menos una operación porque una inversión dentro de b_i nunca puede llevar a una solución no factible, por la factibilidad de la función de vecindad CT. Como el arco $(k, u_{i1}) \in \overline{G}_{\sigma_0}$, los arcos $(h, u_{i1}) \notin \overline{G}_{\sigma_0}$ y $h \in b'$, entonces k esta planificada antes que cualquier $h \in b'$ en $\overline{G}_{\sigma_0}$. De esta forma, la secuencia (j, k, b'), que es una de las producidas por la vecindad CEI, reduce la distancia entre $\overline{G}_\sigma$ y $\overline{G}_{\sigma_0}$. □

Como llevar a cabo la comprobación de factibilidad de forma exacta es muy costoso, se utiliza una condición suficiente (Teorema 3.1.4), que aunque descarta alguna solución factible, son soluciones que no mejoran y se descartan con una probabilidad muy baja. Sin embargo, usar esta condición sacrifica la propiedad de conectividad por lo que se pierde esta garantía en aras de la eficiencia.

Teorema 3.1.4. *Sea x la operación a mover y $b = (b_1, x, b_2)$ el bloque crítico en el que se encuentra donde $b_1 = (b_1', b_1'')$ y $b_2 = (b_2', b_2'')$. El nuevo orden después de la operación de inserción $b' = (b_1', x, b_1'', b_2)$ es factible si que cumple la desigualdad $est_{SJ_k} + p_{SJ_k} > est_{PJ_x} \forall k \in b_1''$ y el orden $b'' = (b_1, b_2', x, b_2'')$ es factible si se cumple $est_{SJ_x} + p_{SJ_x} > est_{PJ_k} \forall k \in b_2$*

Demostración. Se realiza la demostración únicamente para el bloque b' pues el otro caso es análogo. Dado el nuevo bloque b', puede existir un ciclo (y por tanto la solución ser no factible) si y solo si hay un camino de $SJ_k, k \in b_1''$ a PJ_x. Si este camino existe, la operación SJ_k tiene que estar planificada antes que la operación PJ_x y la desigualdad $est_{SJ_k} + p_{SJ_k} \leq est_{PJ_x}$ debe cumplirse. Por tanto, si $\forall k \in b_1''$ no se cumple esta desigualdad, o lo que es lo mismo, se cumple su negación $est_{SJ_k} + p_{SJ_k} > est_{PJ_x}$, la solución es factible. □

3.1.2 Búsqueda tabú

La búsqueda tabú es un algoritmo de búsqueda monoestado introducido por Fred Glover en 1986 [15]. Sigue el esquema básico introducido en el Algoritmo 3.1 al que añade una estructura de memoria adicional, conocida como lista tabú, que le permite almacenar información sobre el proceso de búsqueda para de esta manera poder escapar de óptimos locales. Para poder escapar de un óptimo local es necesario permitir soluciones que no mejoren la actual, pero haciendo esto es fácil entrar en un ciclo. La lista tabú constituye un mecanismo de memoria a corto plazo que permite limitar este comportamiento, aunque solo hasta cierto punto. La estrategia de selección del vecino lo más habitual es que sea siempre el mejor de los que no sean tabú, aunque esta solución no mejore a la actual. En el Algoritmo 3.2 se puede ver el esquema básico de este algoritmo de búsqueda, después en la Sección 3.1.2.1 se detalla el comportamiento de la lista tabú, su componente más importante y, finalmente, en la Sección 3.1.2.2 se describen los detalles más finos que permiten un mejor funcionamiento.

Algoritmo 3.2: Esquema de la búsqueda tabú.

```
Data: P problema a resolver
Result: M mejor solución encontrada
S = ObtenSoluciónInicial(P);
M = S;
while not CriterioParada() do
    N = ObtenVecinos(S);
    R = SeleccionaMejorVecinoNoTabú(N,L);      ▷ Incluye criterio aspiración
    L = ActualizaListaTabú(L,S,R);
    S = R;
    if S > M then
        M=S;
    end
end
return M
```

3.1.2.1 Lista tabú

Ya se ha indicado que la lista tabú es una estructura de memoria que guarda una traza de la búsqueda con el objetivo de guiarla en la dirección adecuada, escapando de óptimos locales y evitando en la medida de lo posible entrar en ciclos.

La primera pregunta que inevitablemente surge es qué se guarda en la lista tabú. Existen muchas opciones dependientes del problema con el que se va a tratar, sin embargo, desde un punto de vista de más alto nivel se pueden agrupar en dos tipos, las que guardan soluciones ya exploradas y las que guardan movimientos realizados. La opción más sencilla es guardar las soluciones, manteniendo un registro de las soluciones ya visitadas se puede evitar que el algoritmo vuelva a ellas y fomentar que explore otras zonas del espacio de estados. No obstante, el espacio de estados es muy grande y muchas soluciones son similares entre ellas tanto en calidad como en sus características, por lo que una lista de este tipo puede resultar altamente ineficiente y su funcionamiento no es siempre tan bueno como se querría. Es lo que tratan de solucionar las listas que almacenan movimientos. Concretamente, en cada iteración de la búsqueda, se almacena el inverso del movimiento que se realizó para pasar de la solución actual a la siguiente. De esta forma, en lugar de evitar volver a soluciones ya vistas, se evita deshacer movimientos realizados recientemente. Los movimientos pueden tener distintos niveles de detalle, siendo posible descomponer los movimientos más grandes en movimientos más pequeños para dar un mayor nivel de precisión en las acciones que se pueden o no hacer. En este trabajo se considerará una lista tabú del segundo tipo, donde los movimientos son la inversión de todas las aristas que sean necesarias para pasar de una solución a otra dentro de una vecindad (en el caso del CT y CET será la inversión de una sóla arista pero en el CEI pueden ser varias).

El segundo factor que más afecta al algoritmo es el tamaño de la lista tabú. Su tamaño no puede ser infinito porque al cabo de unas iteraciones se prohibirían demasiadas soluciones, evitando llegar a estados interesantes, además del evidente excesivo gasto de memoria que se produciría. Existen dos alternativas, listas con tamaño fijo y listas con tamaño variable. En ambos casos el comportamiento es FIFO (*first in first out*), es decir, cuando la lista se llena se sustituye el elemento más antiguo. En las listas de tamaño fijo está claro que siempre se mantiene en memoria el mismo número de elementos, siendo este número dependiente del problema. En las listas de tamaño variable hay más diversidad de comportamiento. Para este trabajo se seguirá la lista dinámica introducida por Dell'Amico y Trubian [8]. En esta estructura se da una cota inferior y una superior del tamaño de la lista, variando el tamaño exacto a lo largo de la ejecución según las siguientes reglas:

- Si la solución encontrada es la mejor hasta el momento, se borra toda la lista, es decir, el tamaño es cero y es como si se reiniciase la búsqueda desde esa solución.
- Si la solución a la que se mueve es peor que la actual, el tamaño aumenta en una unidad siempre y cuando no supere la cota superior.
- Si la solución a la que se mueve es mejor que la actual, el tamaño disminuye en una unidad siempre y cuando no supere la cota inferior.

3.1.2.2 Otros elementos de la búsqueda tabú

Hasta ahora se ha presentado la búsqueda tabú más sencilla, sin embargo se le pueden añadir elementos adicionales para corregir algunos de sus problemas o para potenciar aun más sus virtudes. En esta sección se va a hablar de dos componentes adicionales el criterio de aspiración (Sección 3.1.2.2.1) y la detección de ciclos (Sección 3.1.2.2.2). Además en la Sección 3.1.2.2.3 se comentará como resolver el caso especial en el que todos los movimientos son tabú.

3.1.2.2.1 Criterio de aspiración

Cuando la lisa tabú no guarda soluciones ya visitadas sino movimientos, puede ser demasiado restrictiva a la hora de prohibir soluciones, evitando que el algoritmo se mueva a candidatos prometedores aun no habiendo riesgo de ir a zonas ya exploradas o entrar en un ciclo. Por eso es común añadir un criterio de aspiración que permita al algoritmo realizar movimientos tabú bajo ciertas circunstancias. Aunque el criterio de aspiración puede ser tan complejo como se quiera, el más básico y ampliamente aceptado es ignorar la condición de tabú cuando el candidato mejore a la mejor solución encontrada hasta el momento. De esta forma se evita que un mecanismo que debería ayudar a la búsqueda impida alcanzar una solución mejor.

3.1.2.2.2 Detección de ciclos

Aunque la lista tabú permite la detección de ciclos, su eficacia se ve limitada a su tamaño. Sin embargo, como ya se ha dicho, el tamaño no debe ser excesivamente grande pues se descartarían muchas soluciones potencialmente buenas. Es por ello que es común ayudar a la lista tabú con un mecanismo de detección de ciclos adicional, que se centre en encontrar ciclos de un tamaño mayor.

Una de las ideas más sencillas para detectar un ciclo es comprobar si se ha pasado un cierto número de veces por una solución sin que la mejor solución encontrada haya mejorado.

Para este trabajo, sin embargo, no se ha usado un mecanismo de detección de ciclos adicional pues la búsqueda tabú se va a combinar con un algoritmo evolutivo y el gasto computacional extra requerido no compensa los beneficios obtenidos. No obstante, dado lo común de su utilización, se ha visto necesario al menos dejar constancia de que ha sido considerado.

3.1.2.2.3 Todos los movimientos tabú

Hasta ahora se ha considerado que nada podía fallar pero durante la ejecución del algoritmo podría ocurrir que todos los posibles candidatos fuesen tabú y ninguno cumpliese con el criterio de aspiración. Existen varias alternativas para solucionarlo, la más básica es parar la búsqueda y retornar la mejor solución encontrada hasta el momento. Sin embargo, dependiendo del problema quizá no sea buena idea porque podría no haberse mejorado todo lo posible. Por ello, una solución más razonable es tomar la mejor solución o una de ellas de forma aleatoria. Además es posible que sea conveniente realizar alguna acción sobre la lista tabú como vaciarla total o parcialmente. Para este trabajo simplemente se cogerá la mejor solución y se mantendrá todo el historial de la lista tabú pues no es una situación que aparezca frecuentemente y es preferible no perder la información.

3.2 Búsqueda multiestado

Los algoritmos de búsqueda multiestado, en oposición a los de búsqueda monoestado, consideran más de una solución al mismo tiempo, es decir, trabajan con una población de soluciones y por ello es también común referirse a ellos como algoritmos poblacionales [36].

Los algoritmos multiestado pueden verse como una mejora iterativa de una población de soluciones. Para ello, se parte de una población inicial de soluciones, que sirve como base para generar una nueva población, de modo que la nueva población reemplaza (total o parcialmente) a la antigua. El proceso se repite hasta que se cumple un criterio de parada. Al igual que en los algoritmos monoestado será necesario evaluar la calidad de las soluciones, usando estimaciones si se considera demasiado costoso y extraer los atributos que hacen a esas soluciones buenas. El esquema general se muestra en el Algoritmo 3.3.

Algoritmo 3.3: Esquema de los algoritmos de búsqueda multiestado.

```
Input: P problema a resolver
Output: M mejor solución encontrada
G = ObténPoblaciónInicial(P);
M = ObténMejorSolución(G);
while not CriterioParada() do
    N = ObténNuevaPoblación(G);
    G = SeleccionaSiguienteGeneración(G,N);
    B = ObténMejorSolución(G);
    if B > M then
        M=B;
    end
end
return M
```

En este caso la variedad de algoritmos es mucho mayor y las fases de creación y selección dependen mucho más del algoritmo concreto que se esté considerando. Sin embargo, se pueden agrupar en familias, siendo una de las más conocidas los algoritmos evolutivos (Sección 3.2.1), en los que se centrará este trabajo.

3.2.1 Algoritmos evolutivos

Los algoritmos evolutivos son un subconjunto de los algoritmos poblacionales cuyo funcionamiento se inspira en el proceso de evolución de las especies presente en la naturaleza. Su esquema se encuentra en el Algoritmo 3.4. Los algoritmos evolutivos más conocidos son los algoritmos genéticos, y será su esquema fundamental el que se siga en este trabajo, sin embargo muchos de sus elementos más característicos van a ser sustituidos, por lo que se ha preferido no clasificarlo como tal para evitar posibles confusiones si se consulta otra bibliografía.

Cada una de las fases del algoritmo evolutivo es llevada a cabo por módulos independientes, llamados habitualmente operadores, que pueden ser sustituidos por otros que cumplan la misma función. Los operadores no tienen por qué aplicarse siempre, se pueden aplicar con una determinada probabilidad y en algunos casos deberá ser así para que cumplan con su función

Algoritmo 3.4: Esquema de los algoritmos evolutivos.

```
Input: P problema a resolver
Output: M mejor solución encontrada
G = ObténPoblaciónInicial(P);
M = ObténMejorSolución(G);
while not CriterioParada() do
    R = Selecciona(G) ;                          ▷ Genera parejas
    foreach C in R do
        O = Cruza(C) ;                           ▷ Genera hijos
        O = Muta(O);
        O = Evalúa(O);
        G = Reemplaza(G,C,O);
    end
    B = ObténMejorSolución(G);
    if B > M then
        M=B;
    end
end
return M
```

correctamente. En las secciones siguientes se detallará la codificación utilizada (Sección 3.2.1.1) y los distintos operadores considerados en este trabajo, concretamente, operadores de decodificación/evaluación (Sección 3.2.1.2), operadores de selección (Sección 3.2.1.3), operadores de cruce (Sección 3.2.1.4), operadores de mutación (Sección 3.2.1.5) y operadores de reemplazo (Sección 3.2.1.6).

3.2.1.1 Codificación

Hasta el momento se ha utilizado la representación basada en grafo disyuntivo por ser la más natural, aceptada y la más cómoda para trabajar desde un punto de vista teórico y para definir vecindades. Sin embargo, en los algoritmos evolutivos es más habitual utilizar una codificación basada en secuencias, llamadas cromosomas siguiendo la jerga biológica, donde cada elemento representa un gen. Existen muchos tipos de codificaciones, siendo el primer punto de diferenciación si son binarias o no, es decir, si los genes son solo ceros y unos o pueden tomar valores más complejos. El conjunto de valores que puede tomar un gen se conoce como alelos.

En el caso del JSP la codificación más aceptada son las permutaciones con repetición [3], una generalización del concepto de permutación para trabajar con elementos repetidos. Concretamente, en el JSP, una permutación con repetición se define como un vector en el que cada gen puede tomar como posibles valores los identificadores de los trabajos, de tal forma que cada identificador de trabajo, cada alelo, aparece tantas veces como tareas tiene dicho trabajo y la i-ésima ocurrencia del identificador j se corresponde con la i-ésima tarea del j-ésimo trabajo. De esta forma, la longitud del vector será igual al número de tareas. Esta codificación permite representar todas las soluciones factibles, al mismo tiempo que no puede representar soluciones infactibles, por lo que evitará tener que utilizar procedimientos de comprobación y reparación

de soluciones a la hora de operar con los cromosomas.

Para codificar una solución siguiendo el esquema de las permutaciones con repetición se parte del grafo disyuntivo y se obtiene su orden topológico haciendo uso del Algoritmo 2.1. De esta forma se obtiene una secuencia con todas las operaciones y simplemente hay que sustituir cada tarea con el identificador de su trabajo. La Figura 3.2 ilustra este procedimiento sobre el ejemplo de la Figura 2.4.

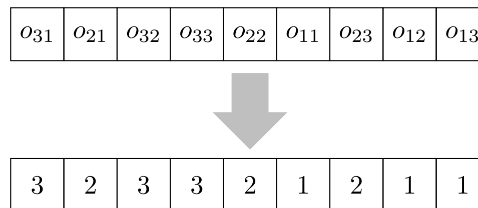

Figura 3.2: Codificación de un orden de tareas como una permutación con repetición.

3.2.1.2 Operadores de decodificación/evaluación

Los operadores de decodificación llevan a cabo el proceso inverso a los de codificación, pasar del cromosoma a la solución. Además, dentro de sus funciones también está evaluar un cromosoma, es decir, obtener la calidad de la solución que se representa, lo que más comúnmente se denomina como su *fitness*.

Recordando la clasificación de las planificaciones introducida en la Sección 2.2, existen distintos decodificadores que, a partir de un cromosoma, pueden generar soluciones en cualquiera de los subconjuntos de soluciones factibles. Dada la codificación presentada anteriormente en la Sección 3.2.1.1 el primer paso está claro que es sustituir cada gen con la tarea dentro del trabajo que se corresponde con la posición relativa del gen dentro de los de su mismo tipo y mantener el orden global dado por el cromosoma, en un proceso inverso al presentado en la Figura 3.2. Si con este orden se aplica el algoritmo Algoritmo 2.2 se obtiene una planificación semiactiva, equivalente a construir el grafo disyuntivo. Sin embargo, para este trabajo el decodificador utilizado será un poco más inteligente, restringiéndose a las soluciones activas, ya que está garantizado que este subconjunto contiene al menos una solución óptima. De esta forma se limita el tamaño del espacio de búsqueda.

El decodificador empleado está basado en un algoritmo más general llamado G&T (Algoritmo 3.5) en honor a sus creadores Giffler y Thompson [14]. Este algoritmo recibe como parámetro la prioridad de las tareas, que en el caso de realizar la decodificación será la posición que ocupan en el cromosoma tal y como se ha hecho anteriormente. Mediante su aplicación se obtiene el tiempo de inicio de las tareas, con el cual es posible calcular tanto el *makespan* como el TWT. Además es posible obtener el grafo disyuntivo orientando las aristas según los tiempos de inicio obtenidos. Nótese que el caso de usar TFN la operación máximo se usará tal y como se ha definido en la Sección 2.4.1.1 pero para calcular cual es el más pequeño se usará el operador relacional introducido en la Sección 2.4.1.2. En la literatura se pueden encontrar otros intentos

de adaptar el algoritmo G&T para ser usados con TFN, pero su rendimiento experimental es similar [30].

Algoritmo 3.5: Algoritmo G&T.

```
Input: A lista con todas las tareas iniciales de cada trabajo, T diccionario con la
       prioridad de cada tarea, P diccionario con el tiempo de procesamiento de cada
       operación, PJ diccionario con el predecesor de cada operación en el trabajo, SJ
       diccionario con el predecesor de cada operación en el trabajo, U diccionario con
       la máquina a la que pertenece cada operación
Output: S diccionario con el EST de cada tarea
▹ Almacenará la última operación planificada en cada máquina
M diccionario indexado con las máquinas;
while A ≠ ∅ do
    ▹ Selecciona la tarea que se puede completar antes
    c = argmin(max(S[PJ[a]] + P[PJ[a]], S[M[U[a]]] + P[M[U[a]]]) + P[a] : a ∈ A);
    ▹ Selecciona las tareas que pueden empezar antes en la misma máquina
      (conflict set)
    t = max(S[PJ[c]] + P[PJ[c]], S[M[U[c]]] + P[M[U[c]]]) + P[c];
    E ={U[a] = U[c] ∧ max(S[PJ[a]] + P[PJ[a]], S[M[U[a]]] + P[M[U[a]]]) < t : a ∈ A};
    c = argmin(T[e] : e ∈E) ;            ▹ Selecciona la tarea más prioritaria
    S[c] = max(S[PJ[c]] + P[PJ[c]], S[M[U[c]]] + P[M[U[c]]]);
    M[U[c]] = c;
    A = A \ c;
    A = A ∪ SJ[c] ;                      ▹ Añade el sucesor en el trabajo si existe
end
return S
```

El algoritmo en el proceso de decodificación cambia el orden de las tareas dado por el cromosoma. Este nuevo orden debe ser trasladado de vuelta al cromosoma, pues en otro caso no habría una correspondencia entre el *fitness* y los genes y el proceso evolutivo no funcionaría correctamente. Esta modificación del cromosoma como resultado del proceso de decodificación y evaluación se conoce como lamarckismo en la literatura.

3.2.1.3 Operadores de selección

El objetivo de los operadores de selección es escoger los mejores individuos para reproducirse, de tal manera que los mejores genes pasen a la siguiente generación y se puedan encontrar soluciones con un mejor *fitness* que las actuales.

3.2.1.3.1 Ruleta

El operador de selección ruleta, *roulette wheel selection* (RWS), es un operador de selección proporcional al *fitness* de los individuos. Su funcionamiento se basa en asignar una probabilidad a cada individuo y escoger uno de forma aleatoria atendiendo a dicha probabilidad. Sea f_i el *fitness* del individuo i, entonces la probabilidad de ser seleccionado p_i viene dada por la Ecuación 3.1.

$$p_i = \frac{f_i}{\sum_j^n f_j} \tag{3.1}$$

Se seleccionan de forma independiente tantos individuos como sean necesarios siguiendo esta misma distribución de probabilidad. Los individuos con un *fitness* más alto tendrán una probabilidad más alta de ser seleccionados, aunque no hay garantía de que sean finalmente seleccionados.

Este operador de selección tiene el problema de que si algunos individuos son muy buenos en comparación al resto, les otorga una probabilidad muy superior, pudiendo provocar una pérdida de diversidad en la población y una convergencia prematura del algoritmo. Por otro lado, si todos los individuos tienen un *fitness* muy parecido, puede no introducir una presión selectiva suficientemente alta como para escoger los mejores individuos. En otras palabras, no funciona bien en casos extremos.

3.2.1.3.2 Parejas

El operador de selección por parejas simplemente divide la población completa en subconjuntos de dos elementos de manera aleatoria dando de esta forma oportunidad de reproducirse a todos los individuos. Este operador de selección no genera ningún tipo de presión selectiva por lo que solo resulta recomendable si la presión se introduce en otro de los operadores, fundamentalmente en el operador de reemplazo.

3.2.1.4 Operadores de cruce

El objetivo de los operadores de cruce es combinar las mejores características de dos individuos padre para generar descendientes, habitualmente dos, que se espera tengan un mejor *fitness*. El tipo de los operadores de cruce que se puedan utilizar dependerá de la codificación utilizada, en este caso la permutación con repetición presentada en la Sección 3.2.1.1. Su probabilidad de aplicación debe ser alta para que los padres se reproduzcan y generen nuevos individuos, aunque también es interesante que de vez en cuando los padres pasen tal cual a la siguiente generación.

3.2.1.4.1 GOX

El operador de cruce *generalized order crossover* (GOX) introducido por Bierwirth [3] es, como su nombre indica, una generalización del operador *order crossover* (OX). Su funcionamiento se ilustrará tomando como padres los cromosomas de la Figura 3.3. Debido a que los genes van a recibir un índice, que representa la posición relativa que ocuparía la tarea dentro del trabajo, los alelos van a ser letras en lugar de números para evitar confusiones.

Para obtener un hijo, uno de los padres actúa como receptor y el otro como donante. Cambiando los roles es posible obtener un segundo hijo. Se escoge una subsecuencia, denominada implante, de entre un tercio y un medio de la longitud total del donante de forma aleatoria. Esta subsecuencia, aunque continua, puede extenderse por los dos extremos, es decir, es como si los extremos estuvieran conectados. En función de si está totalmente dentro o se extiende por los bordes se procederá de forma diferente:

Padre 1

Genes	B	A	B	B	C	A	C	C	B	A
Índices	1	1	2	3	1	2	2	3	4	3

Padre 2

Genes	A	B	B	A	C	A	B	C	B	C
Índices	1	1	2	2	1	3	3	2	4	3

Figura 3.3: Cromosomas para mostrar el comportamiento del operador de cruce GOX.

- En el caso de que recaiga completamente dentro, los genes del donante se insertan en el cromosoma del receptor delante del gen cuya pareja alelo-índice coincide con la del primer gen del implante. Posteriormente, todos los genes en el receptor que tengan la misma pareja alelo-índice que en el implante son eliminados. En la Figura 3.4 se representa la situación, el implante aparece sombreado en verde y los genes a borrar en rojo. En este caso los índices del implante no siempre se van a mantener en el cromosoma hijo, sin embargo la posición elegida para la inserción hace que no ocurra frecuentemente.

Padre 2 - Donante

Genes	A	B	B	A	C	A	B	C	B	C
Índices	1	1	2	**2**	**1**	**3**	**3**	2	4	3

Padre 1 - Receptor

Genes	B	A	B	B	C	A	A	C	A	B	C	C	B	A
Índices	1	1	2	**3**	**1**	**2**	**2**	**1**	**3**	**3**	2	3	4	**3**

Hijo

Genes	B	A	B	A	C	A	B	C	C	B
Índices	1	1	2	2	1	3	3	2	3	4

Figura 3.4: Operador cruce GOX en el caso de que el implante recaiga dentro.

- En el caso de que se extienda por los bordes, se inserta cada extremo a cada lado del receptor y se borran aquellos genes que tengan la misma pareja alelo-índice que en el implante. En la Figura 3.5 se representa este caso. Se usa el mismo código de colores que en el caso anterior.

3.2.1.5 Operadores de mutación

Los operadores de mutación actúan de forma individual sobre los individuos, produciendo pequeños cambios aleatorios sobre los mismos, con el objetivo de que aparezcan nuevas características interesantes y así aumentar la diversidad. Su probabilidad debe ser baja pues en otro caso la búsqueda se diversificaría demasiado y no llegaría a converger, convirtiéndose en una búsqueda prácticamente aleatoria.

Padre 2 - Donante

Genes	A	B	B	A	C	A	B	C	B	C
Índices	**1**	**1**	2	2	1	3	3	2	**4**	**3**

Padre 1 - Receptor

Genes	A	B	B	A	B	B	C	A	C	C	B	A	B	C
Índices	**1**	**1**	**1**	**1**	2	3	1	2	2	3	**4**	**3**	**4**	**3**

Hijo

Genes	A	B	B	B	C	A	C	A	B	C
Índices	1	1	2	3	1	2	2	3	4	3

Figura 3.5: Operador cruce GOX en el caso de que el implante se extienda por los bordes.

3.2.1.5.1 Intercambio

Dada la codificación escogida, el operador de mutación más natural y fácil de implementar es el intercambio de la posición entre dos genes de forma aleatoria. El cromosoma resultante seguirá siendo factible pues está garantizado por la propia codificación.

3.2.1.6 Operadores de reemplazo

Los operadores de reemplazo establecen como, a partir de la generación anterior y la nueva calculada, escoger una nueva población para ser usada en la siguiente iteración.

3.2.1.6.1 Generacional

El operador de reemplazo generacional es el más utilizado, y como su nombre indica simplemente introduce en la siguiente generación los hijos, descartando a los padres.

3.2.1.6.2 Torneo

El operador de reemplazo basado en torneo introduce en la siguiente generación las dos mejores soluciones a escoger entre los dos padres y los dos hijos. Este operador de reemplazo genera una presión selectiva muy alta por lo que no es adecuado con operadores de selección que también la aplican como RWS.

3.2.1.6.3 Elitismo

Aunque se introduce en esta sección, el elitismo no es un operador de reemplazo propiamente dicho, sino que es un complemento a cualquier otro operador de reemplazo, comúnmente el generacional. Su función es simplemente garantizar que la mejor solución de la generación actual pase a la siguiente, evitando que se pierda, lo cual, dependiendo de la selección de operadores que se haga, puede no estar garantizado. Aunque no es lo más común, el concepto de elitismo se puede generalizar, asegurando que pase a la siguiente generación un grupo de soluciones.

3.3 Algoritmos híbridos

Los algoritmos híbridos tratan de aprovechar las sinergias entre esquemas de búsqueda con características diferentes para crear un algoritmo que aúne las mejores características de todos ellos permitiendo encontrar mejores soluciones.

Este trabajo solo va a considerar los algoritmos mémeticos, que combinan los algoritmos evolutivos, con un algoritmo de búsqueda monoestado.

3.3.1 Algoritmos meméticos

Los algoritmos meméticos siguen el mismo esquema básico que los algoritmos evolutivos, con la salvedad de que se añade una nueva fase de mejora. Esta fase consiste en decodificar los cromosomas hijo de forma adecuada y usarlos como punto inicial para un algoritmo monoestado. Al igual que otros operadores, el operador de mejora puede tener asignada una cierta probabilidad, evitando se produzca siempre si es muy costoso. De esta manera se consigue mejorar los nuevos cromosomas, potenciando sus mejores características. Hay que tener en cuenta, que al igual que con los operadores de decodificación, los cromosomas pueden ser alterados ya que el orden de las operaciones puede cambiar (ahora de hecho es mucho más probable y se espera que sea así puesto que indicaría que la solución ha sido mejorada). En particular, en este trabajo los cromosomas se decodificarán como grafos disyuntivos que servirán como punto de partida para la búsqueda tabú descrita en la Sección 3.1.2. En el Algoritmo 3.6 se muestra el esquema de los algoritmos meméticos.

Algoritmo 3.6: Esquema de los algoritmos meméticos.

```
Input: P problema a resolver
Output: M mejor solución encontrada
G = ObténPoblaciónInicial(P);
M = ObténMejorSolución(G);
while not CriterioParada() do
    R = Selecciona(G) ;                          ▷ Genera parejas
    foreach C in R do
        O = Cruza(C) ;                           ▷ Genera hijos
        O = Muta(O);
        O = Decodifica(O);
        O = Mejora(O);
        G = Reemplaza(G,C,O);
    end
    B = ObténMejorSolución(G);
    if B > M then
        M=B;
    end
end
return M
```

CAPÍTULO 4

Desarrollo software

En este capítulo se detallarán las decisiones tomadas a la hora de desarrollar el *software* realizado para llevar a cabo los experimentos. Cabe notar que, en muchos aspectos, este *software* se desvía de lo que se podría considerar un proyecto *software* estándar. Esto se debe a la naturaleza del proyecto, más enfocado a la investigación y por tanto más centrado en el modelado del problema y el diseño de los algoritmos.

Serán cinco puntos a los que se dedique atención: los requisitos (Sección 4.1), es decir, los objetivos que debe satisfacer el software; la metodología (Sección 4.2), que explica cómo se ha planeado el proceso de desarrollo; el diseño (Sección 4.3), que detalla cómo se han organizado los diferentes componentes desarrollados; las pruebas (Sección 4.4), que tratan de garantizar la corrección y la documentación (Sección 4.5), que explica como se usa el *software* y facilita la mantenibilidad.

4.1 Requisitos

En esta sección se detallarán los requisitos funcionales (Sección 4.1.1), aquellos que definen la funcionalidad que debe tener el *software* y no funcionales (Sección 4.1.2), relativos a las propiedades que debe cumplir.

4.1.1 Requisitos funcionales

1. Requisitos de problemas

    1.1. El *software* podrá resolver el problema del JSP minimizado el *makespan* con tiempos de procesamiento deterministas.

    1.2. El *software* podrá resolver el problema del JSP minimizado el *makespan* con tiempos de procesamiento difusos.

    1.3. El *software* podrá resolver el problema del JSP minimizado el TWT con tiempos de procesamiento deterministas.

    1.4. El *software* podrá resolver el problema del JSP minimizado el TWT con tiempos de procesamiento difusos.

2. Requisitos de metaheurísticas

   2.1. El *software* podrá resolver los problemas haciendo uso de un algoritmo de búsqueda tabú.

   2.2. El *software* podrá resolver los problemas haciendo uso de un algoritmo evolutivo.

   2.3. El *software* podrá resolver los problemas haciendo uso de un algoritmo memético.

4.1.2 Requisitos no funcionales

1. Requisitos de mantenibilidad

   1.1. El *software* deberá desarrollarse como una librería, es decir, de forma modular de tal forma que sea sencillo sustituir elementos individuales por otros que cumplan la misma función haciendo uso de un algoritmo distinto.

2. Requisitos de rendimiento

   2.1. La lógica principal del *software* deberá estar escrita en C++ siguiendo el estándar C++17.

3. Requisitos de portabilidad

   3.1. El *software* deberá funcionar sobre Linux (kernel 5.0 o superior) por lo que solo se podrán usar librerías que funcionen sobre dicho sistema operativo.

4.2 Metodología

Para el desarrollo del *software* se ha seguido una metodología iterativo incremental [33]. De esta manera, en cada etapa, se han ido generando versiones funcionales del programa que daban solución a un determinado subconjunto de los requisitos. El programa obtenido al finalizar cada una de las fases era completamente funcional, de modo que aunque no daba soporte a todos los requisitos, permitía realizar una preevaluación de los objetivos. Con el conocimiento adquirido en cada una de las etapas, se realizaron pequeños ajustes al diseño, consiguiendo mejorar la calidad del resultado final. El desarrollo completo se dividió en cuatro etapas:

1. Resolución del JSP clásico haciendo uso de una búsqueda tabú. En esta primera etapa se obtuvieron unos resultados preliminares para confirmar que la implementación del modelado del problema era correcta y la eficiencia era la esperada. Se dio cobertura por tanto a los requisitos funcionales 1.1 y 2.1.

2. Resolución del JSP clásico haciendo uso de un algoritmo memético, añadiendo de esta forma el componente evolutivo. De esta forma ya estaban implementados todos los algoritmos considerados y podía evaluarse la calidad de la metaheurística seleccionada. Esta fase cubrió los requisitos funcionales 2.2 y 2.3.

3. Resolución del JSPTWT con todos los algoritmos implementados. En este paso se dejaron implementadas todas las funciones objetivo consideradas, permitiendo confirmar que el comportamiento era bueno también cuando se quería optimizar otra medida. En esta fase se cubrió el requisito funcional 1.3.

4. Resolución del JSP para ambas funciones objetivo haciendo uso de tiempos de procesamiento difuso. De esta manera se dio cobertura a todos los requisitos definidos en la Sección 4.1, dando por finalizado el proceso de desarrollo. Los requisitos funcionales que se cubrieron en esta fase fueron el 1.2 y 1.4.

4.3 Diseño

Durante esta sección se detallará el diseño realizado. Atendiendo a los requisitos presentados en la Sección 4.1 el software deberá desarrollarse de forma modular en el lenguaje de programación C++ siguiendo su versión C++17 [20]. La forma más natural para conseguirlo es hacer uso de plantillas siguiendo el estilo de la librería estándar de dicho lenguaje [34]. Este paradigma de programación recibe el nombre de programación genérica y permite un muy bajo acoplamiento entre las clases.

De esta manera, se puede mantener de forma separada la lógica de los algoritmos y la de los problemas, no sabiendo los algoritmos nada de la implementación de los problemas que van a resolver y permitiendo trabajar de forma abstracta. Los problemas serán los encargados de implementar las funciones necesarias para dar respuesta a los requisitos de los algoritmos. Será responsabilidad del programa principal asociar los problemas con los algoritmos para encontrar la solución.

Un problema que irremediablemente surge al utilizar este paradigma es cómo definir las interfaces que deben implementar las clases para garantizar su correcto funcionamiento. Desafortunadamente, en C++ no existe el concepto de interfaz como tal, sí presente en otros lenguajes de programación. Una posible solución sería el uso de la herencia, derivando todas las clases que presentan un comportamiento común de la misma clase base, que sería declarada como abstracta con la obligatoriedad de redefinir todos sus métodos, que harían las veces de interfaz. Sin embargo, esta solución, que es común aplicarla para resolver situaciones similares, no es viable en este caso porque la herencia es un mecanismo de tiempo de ejecución y no permite una buena integración con las plantillas que se resuelven durante el proceso de compilación. Evidentemente esto supone un problema importante para aplicaciones que requieren una gran generalización y flexibilidad, como la que se está tratando. El comité encargado de definir el estándar de C++ se ha dado cuenta de las limitaciones y en el futuro estándar C++20 serán introducidos los *concepts* que dan solución a esta problemática [19]. No obstante este estándar todavía no está aprobado, su soporte en los compiladores es limitado y no es la versión definida en los requisitos. Por esta razón, las interfaces quedarán únicamente indicadas como parte de la documentación y no serán parte del código.

Además, como el software ha sido concebido de forma modular, se ha preferido el uso de iteradores sobre contenedores específicos para realizar la comunicación entre los componentes.

De esta manera los distintos componentes pueden comunicarse entre ellos de forma cómoda y segura, siempre manteniendo el control sobre los datos a los que se accede, sin necesidad de transformaciones y evitando las costosas copias. De nuevo, esta forma de programación viene inspirada por la librería estándar del lenguaje.

La forma más aceptada de realizar el modelado es haciendo uso del lenguaje UML (*Unified Modeling Language*) [26]. Sin embargo, para este caso no se ha podido hacer uso del mismo debido a sus limitaciones. Aunque en el estándar UML se establecen semánticas para el uso de plantillas, su uso no está muy extendido por lo sobrecargado de la sintaxis que utiliza; además, en cualquier caso, no permite modelar toda la riqueza que ofrece C++. No hay que olvidar que UML está diseñado para dar soporte al paradigma de programación orientado a objetos y, aunque el paradigma de programación genérica aquí utilizado es bastante similar, existen diferencias fundamentales entre ambos. Es por ello que para la realización de los diagramas se ha optado por un pseudo-UML utilizado por Doxygen [9], el estándar de facto para el desarrollo de documentación en C++ y otros lenguajes que carecen de mecanismos nativos. El diagrama de la estructura completa de programa se puede encontrar en `https://pablogarciagomez.github.io/jsp/diagrams/`.

Todo el código se encuentra publicado en `https://github.com/pablogarciagomez/jsp`. Para hacer patente la división entre los problemas y los algoritmos, el código está separado en los directorios `problems` y `metaheuristics` de forma respectiva.

4.4 Pruebas

La mayoría de los algoritmos implementados son algoritmos estocásticos, es decir, su comportamiento es no determinista y no es posible predecir su salida conociendo los argumentos de entrada. Sin embargo, a la hora de programar, se utilizan números pseudoaleatorios, cuyos valores dependen de un valor inicial conocido como semilla. Así, si se fija la semilla, es posible conocer la secuencia de valores que se utilizan en las partes estocásticas y predecir la salida a la hora de realizar pruebas. Para conseguirlo todos los métodos que requieren de un generador de números aleatorios lo reciben como parámetro, de esta forma en dos ejecuciones distintas es posible obtener el mismo resultado permitiendo comparar los cambios realizados. Esto resulta especialmente útil cuando se añaden optimizaciones a los algoritmos que no deben cambiar su comportamiento pero si su rendimiento. Si el resultado entre la versión vieja y la nueva difiere es que algo se ha hecho mal.

Habiendo desarrollado el software de esta manera sería posible la implementación de pruebas unitarias para comprobar la corrección de cada uno de los componentes. Para ello habría que crear generadores de números aleatorios con un comportamiento determinista y conocido y pasárselos como argumento. Estos objetos falsos, que después no se usarán de verdad, puesto que al no ser suficientemente aleatorios los resultados obtenidos estarían desviados, reciben comúmente el nombre de *mocks*.

No obstante, esto es mucho más complicado de lo que parece a primera vista. Aunque se conoce la secuencia de números aleatorios, estos serán usados en la mayoría de los casos por otros algoritmos pertenecientes a librerías externas, que encadenados al propio código

implementado hacen inviable la predición del resultado. Es decir, aunque es posible escribir pruebas unitarias y en el caso de una aplicación crítica serían necesarias, para este software están fuera de lugar por la profundidad que se requiere.

Aun así sigue siendo necesario obtener alguna confirmación de que el software desarrollado cumple con su función. Para conseguirlo el enfoque seguido ha sido en lugar de comprobar cada parte de forma aislada, realizar los test sobre las metaheurísticas y ver su comportamiento. El primer paso es la ejecución del software un número suficientemente grande de veces; concretamente se ejecutó 30 veces que es considerado en estadística un número a partir del cual una muestra se considera representativa. Si el programa es correcto, y dada la naturaleza de los algoritmos, la calidad de las soluciones que se obtenga en n ejecuciones deberá converger hacia una distribución normal. Si no ocurre puede deberse a dos razones, una mala configuración de los parámetros del algoritmo, o bien que el algoritmo está mal implementado. Para descartar la primera basta seguir las directrices que se han dado para ajustar los algoritmos, buscando un equilibrio entre diversificación e intensificación. Si el algoritmo es bueno o es malo dependerá de los valores que se obtengan, aunque se detallará más este aspecto en el Capítulo 5.

Existen otros elementos, que aun no siendo estocásticos son demasiado complejos como para que los test unitarios sean efectivos, pues haría falta un número muy elevado de ellos para obtener una buena cobertura. Para estos algoritmos se puede llevar a cabo el mismo procedimiento, ejecutar un número suficientemente elevado de entradas aleatorias y asegurarse de que convergen a una distribución normal.

Evidentemente las garantías obtenidas con este tipo de pruebas son distintas que con las pruebas unitarias, pues el algoritmo podría tener un buen comportamiento aun no realizando la función que se espera de él. Sin embargo, se consideran suficientes, pues aun con los posibles problemas el algoritmo habría demostrado el desempeño exigido.

4.5 Documentación

Como se ha dicho, para escribir la documentación se ha usado Doxygen [9], el estándar de facto para el desarrollo de documentación en C++. De esta manera es posible exportar toda la documentación a formato web para su fácil consulta o a formato LaTeX para la creación de un manual.

Dado que el software desarrollado se ha organizado como una librería, la documentación cobra una especial relevancia, ya que detalla toda la funcionalidad ofrecida permitiendo su reutilización en trabajos posteriores.

La documentación se encuentra publicada en `https://pablogarciagomez.github.io/jsp/doc/`.

Capítulo 5

Resultados experimentales

En este capítulo se recogerán los resultados experimentales obtenidos con los algoritmos presentados. A lo largo del trabajo se han propuesto diferentes combinaciones de problemas, funciones objetivo y metaheurísticas para darles solución. Aquí solo se mostrarán los resultados para las variantes más interesantes tal y como se explica a continuación.

Desde que se empezó a trabajar con el JSP varios autores han propuesto conjuntos de instancias conocidos como *benchmarks* sobre los que evaluar los algoritmos. Para este trabajo, se considerarán aquellas instancias que han sido identificadas como más díficiles para el JSP clásico, el único para el cual existe un estudio [1]. Concretamente estas instancias son `la21`, `la24`, `la25` ($n = 15, m = 10$), `la27`, `la29` ($n = 20, m = 10$), `la38`, `la40` ($n = 15, m = 15$), `abz7`, `abz8` y `abz9` ($n = 20, m = 15$). A estas instancias se les sumará el `ft10` ($n = 10, m = 10$) y `ft20` ($n = 20, m = 5$) de Fisher y Thompson [28]. Las instancias están propuestas para el JSP clásico por lo que deberán ser adaptadas en función del problema que se vaya a resolver, posteriormente se detallará como.

En cuanto a algoritmos se considerará únicamente un algoritmo memético porque combina los puntos positivos de todos los presentados. La metaheurística monoestado con la que se combinará será la búsqueda tabú. En cuanto a las funciones de vecindad, como el algoritmo mémetico es bastante costoso, se optará por el CET, puesto que es el más pequeño de los vistos y experimentalmente no se ha visto que rinda significativamente peor. La lista tabú a utilizar tendrá un tamaño dinámico pudiendo variar en el rango $[n+m, 2(n+m)]$ donde n es el número de trabajos y m el número de máquinas siguiendo la notación ya usada. Su criterio de parada serán $2n+m$ iteraciones sin encontrar una mejor solución. La parte evolutiva utilizará la selección por parejas que, como no produce presión selectiva, se combinará con el reemplazo por torneo. El operador de cruce será el GOX y el de mutación el intercambio. No se incluirá elitismo puesto que el reemplazo por torneo cumple una función similar, dejando pasar solo a los mejores, y usar los dos componentes de forma conjunta llevaría a una convergencia muy prematura. Los operadores de cruce y reemplazo se ejecutarán incondicionalmente, mientras que el de mutación tendrá una probabilidad de 0.1. La fase de búsqueda monoestado se ejecutará sobre todos los individuos. En cuanto al tamaño de la población será $n \cdot m$. El criterio de parada serán n iteraciones sin encontrar una solución que mejore o hasta que la calidad media de la población

sea igual a la de la mejor solución conocida, en cuyo caso no tiene sentido seguir ejecutando en espera de una mutación.

Como se puede apreciar, se ha hecho depender los paramétros de configuración del tamaño de los problemas, de tal manera que el gasto computacional se adapte a la dificultad; además se ha dado más importancia al número de trabajos que al de máquinas pues durante las pruebas preliminares se ha observado que tiene una mayor relevancia. Para escoger los parámetros simplemente se ha realizado un ajuste experimental siguiendo una estrategia secuencial de optimización [36].

Para cada una de las instancias de prueba se ejecutará el algoritmo 30 veces por ser este un número que en estadística está tradicionalmente considerado como mínimo para el tamaño de una muestra. De esta manera, entre todos los resultados obtenidos se escogerá el mejor y se calculará la media y la desviación típica. Aunque lo más interesante es la mejor solución encontrada, puesto que será la solución que se retorne como resultado, la media aporta información sobre lo estable de las soluciones obtenidas. Además, la desviación típica ofrece información sobre la dispersión de las soluciones. No hay que olvidar que los algoritmos que se están contemplando son algoritmos estocásticos, es decir, su ejecución es no determinista, y se deben ejecutar varias veces para eliminar el factor de “mala suerte”.

Una vez aclarados los puntos comunes, en las siguientes secciones se mostrarán los resultados obtenidos, la Sección 5.1 tratará el JSP minimizando el TWT con tiempos de procesado deterministas, en la Sección 5.2 se evaluará el JSP minimizando el *makespan* con tiempos de procesado difusos y por último, en la Sección 5.3, se trabajará con el JSP minimizando el TWT con tiempos de procesamiento difusos. Además, en el Apéndice A se realiza un estudio de sinergia para comprobar de forma experimental la cooperación entre el algoritmo evolutivo y la búsqueda tabú.

5.1 Estudio experimental del JSP minimizando el TWT con tiempos de procesamiento deterministas

En esta sección se va a considerar el JSP minimizando el TWT con tiempos de procesado deterministas. Para este problema las mejores soluciones conocidas hasta el momento están recogidas en [2] y serán con las cuales se realice la comparación siempre que sea posible, ya que el conjunto de instancias consideradas no coincide totalmente.

Evidentemente, para poder realizar una comparación, las instancias originales deben ser adaptadas de la misma manera. El primer paso es asignar a cada trabajo una fecha límite; esta fecha se calcula como $d_j = f \sum_{k=1}^{m_j} p_{jk}, 1 \leq j \leq n$, donde f puede tomar tres valores $f \in \{1.3, 1.5, 1.6\}$ que dan lugar a fechas límite cada vez más ajustadas. Los pesos de los trabajos se establecen de la siguiente manera: el primer 20 % de los trabajos recibe peso 4, el siguiente 60 % recibe peso 2 y el último 20 % recibe peso 1. De esta manera, por cada instancia para el JSP clásico, se obtienen tres instancias para el JSPTWT.

Dicho esto, los resultados obtenidos se pueden ver en la Tabla 5.1. La tabla consta de tres grupos, uno por cada factor de los propuestos. Las columnas denominadas BKS hacen referencia

a la mejor solución conocida en la literatura, y las siglas vienen del inglés *best known solution*. No existe ninguna garantía de que estas soluciones sean óptimas, prueba de ello es que las instancias la27, la29 y la38, todas con el factor 1.5, han sido mejoradas en este trabajo. Estas soluciones, que ahora pasan a ser las mejores, se han destacado en negrita. Para realizar la comparación con los resultados obtenidos, se calcula el error relativo como $RE = \frac{B-I}{B}$ donde B e I son respectivamente el TWT de la mejor solución conocida y de la solución a evaluar. Esta forma de calcular el error tiene la desventaja de que no puede ser calculado cuando el mejor valor conocido es cero. Una solución propuesta en la literatura es en lugar de calcular el error relativo por cada instancia, calcular un único error global, según la expresión $RE_{glob} = \frac{\sum_i^T B_i - \sum_i^T I_i}{\sum_i^T B_i}$ donde T es el número de instancias que se está evaluando [7]. Sin embargo, haciendo esto, los resultados con un orden de magnitud más pequeño quedan enmascarados por aquellos más grandes. Por esta razón se ha decidido no utilizar esta técnica, más aún cuando para este caso concreto el problema solo surge para una instancia en la comparación con el valor medio.

Analizando los resultados, se puede observar que el algoritmo implementado es altamente competitivo, obteniendo un error relativo medio para la mejor solución encontrada inferior al 5 %. Esto cobra especial relevancia al tratarse de un conjunto de instancias seleccionadas por su mayor dificultad y porque las comparaciones se realizan con los mejores métodos existentes en la literatura. Además se han encontrado tres soluciones que mejoran las mejores hasta el momento y se han igualado los resultados en otros cinco casos.

5.2 Estudio experimental del JSP minimizando el makespan con tiempos de procesamiento difusos

En esta sección se va a trabajar con el JSP minimizando el *makespan* con tiempos de procesado difusos. El primer paso es adaptar las instancias, que en este caso es convertir los números deterministas de las instancias originales en números difusos. El método seguido es el propuesto en [11] adaptado para trabajar con TFN según [16]. Dado un tiempo de procesamiento determinista x, un número difuso triangular simétrico $A(x)$ se puede obtener asignando el valor central $a^2 = x$, escogiendo el valor inferior a^1 de forma aleatoria en el rango $[0.85x, x]$ y tomando el valor superior $a^3 = 2a^2 - a^1$. Esta forma de calcular los TFN asegura que el valor óptimo para el JSP determinista sea una cota inferior para el difuso. Nótese que, al ser un procedimiento no determinista, da lugar a diferentes instancias difusas en cada ejecución. Afortunadamente, existen las instancias del artículo [21] que según [31] es el estado del arte para este problema, por lo que se puede hacer la comparación con ellas.

Los resultados obtenidos se pueden ver en la Tabla 5.2. El formato es el mismo que en la tabla anterior. Los resultados mostrados son los del *makespan* esperado pues es el que se ha usado para ordenar las soluciones.

Los resultados obtenidos mejoran la mejor solución encontrada hasta el momento en la mayoría de los casos, y en el resto la igualan. Aunque la mejora no es muy grande porcentualmente, un 0.29 % de media, sí que es una mejora importante porque afecta a muchas de las instancias. Además, en cualquier caso, la mejora no podría ser muy grande, porque las mejores soluciones

Tabla 5.2: Resultados del JSP minimizando el makespan con tiempos de procesamiento difusos.

Instancia	Cota inferior	BKS	Makespan esperado			RE (%)	
			Mejor	Medio	SD	Mejor	Medio
ft10	930	932.75	932.75	934.95	2.59	0.00	0.24
ft20	1165	1165.50	1165.50	1189.64	32.35	0.00	2.07
abz7	656	670.50	**668.00**	669.63	1.26	-0.37	-0.13
abz8	645	682.75	**673.00**	681.32	3.01	-1.43	-0.21
abz9	661	693.25	**687.00**	689.09	1.28	-0.90	-0.60
la21	1046	1051.25	**1049.50**	1057.35	7.56	-0.17	0.58
la24	935	940.75	**940.25**	941.12	0.78	-0.05	0.04
la25	977	978.50	978.50	979.10	0.66	0.00	0.06
la27	1235	1239.75	**1237.00**	1253.68	5.45	-0.22	1.12
la29	1152	1168.75	**1167.50**	1176.43	11.00	-0.11	0.66
la38	1196	1203.25	**1201.00**	1203.60	1.33	-0.19	0.03
la40	1222	1229.75	1229.75	1232.84	5.95	0.00	0.25
						-0.29	0.34

hasta el momento ya estaban muy cerca de la cota inferior, la cual no puede ser traspasada. Hay que destacar que el algoritmo tiene un comportamiento muy estable, siendo el error relativo para el caso medio inferior al 0.5 % y la desviación típica muy pequeña en comparación con el *makespan* medio.

5.3 Estudio experimental del JSP minimizando el TWT con tiempos de procesamiento difusos

En esta sección se va a tratar el JSP minimizando el TWT con tiempos de procesamiento difusos, un problema para el cual no se ha encontrado ninguna referencia en la que se haya intentado resolver con anterioridad, por lo que no hay ninguna solución con la que comparar el funcionamiento del algoritmo. Tratar de obtener las soluciones óptimas con un algoritmo exacto está completamente fuera de lugar pues como se ha dicho es un problema *NP-Hard* e incluso para el JSP clásico el tiempo necesario es tan sumamente elevado que se hace inviable. No obstante, durante las secciones anteriores ya se ha demostrado el buen desempeño del algoritmo, por lo que se pueden extrapolar las conclusiones. Es decir, se considera la minimación del TWT con tiempos de procesado deterministas como un caso especial del TWT con tiempos de procesado difusos en el que todas las componentes de los TFN tienen el mismo valor. Además, se ha comprobado que el algoritmo también ofrece buenos resultados para el *makespan* con tiempos de procesamiento difusos. Habiendo evaluado la calidad de la metaheurística, se ejecuta por primera vez para el TWT con tiempos de procesado difusos, obteniendo unos resultados que sirvan de referencia para futuros experimentos. La extrapolación no tiene por que ser correcta, que el algoritmo funcione bien sobre un problema no da ninguna garantía para que funcione bien sobre otro, sin embargo, en este caso básicamente se está atacando el mismo problema, con variaciones en la función objetivo y la representación de los números, que son combinaciones

de los dos casos anteriores.

Como se hizo anteriormente, el primer paso es adaptar las instancias de prueba. Debido a la ausencia de referencias, estas instancias será la primera vez que se proponen. Para construirlas se partirá de las instancias propuestas en la Sección 5.2 y se calculará la fecha de entrega haciendo uso del procedimiento descrito en la Sección 5.1, utilizando únicamente el valor de la componente central de los TFN puesto que las fechas límite consideradas son deterministas. Según este procedimiento, el TWT óptimo para las instancias deterministas serviría de cota inferior para estas, sin embargo, como no se conoce, la única forma de ver que el algoritmo funciona correctamente es que los resultados obtenidos se queden cerca de los obtenidos en el caso determinista.

Los resultados se encuentran en la Tabla 5.3 que comparte el mismo formato que las anteriores. Al igual que antes, el TWT mostrado es el esperado pues es el que se usa para ordenar las soluciones. Únicamente comentar que los resultados efectivamente se quedan muy cerca de los obtenidos en la Sección 5.1 para el TWT con tiempos de procesado deterministas por lo que es de esperar que estén cerca de los valores óptimos.

Tabla 5.1: Resultados del JSP minimizando el TWT con tiempos de procesamiento deterministas.

Instancia	f=1.3						f=1.5						f=1.6					
	BKS	TWT			RE (%)		BKS	TWT			RE (%)		BKS	TWT			RE (%)	
		Mejor	Medio	SD	Mejor	Medio		Mejor	Medio	SD	Mejor	Medio		Mejor	Medio	SD	Mejor	Medio
ft10	1363	1363	1504.87	163.05	0.00	10.41	394	394	420.27	29.18	0.00	6.67	141	148	176.37	11.63	4.96	25.08
ft20	-	13233	13715.07	293.88	-	-	-	11267	11703.60	281.13	-	-	-	10256	10765.57	239.86	-	-
abz7	-	3319	3530.00	140.73	-	-	-	1279	1344.37	46.71	-	-	-	609	698.70	64.96	-	-
abz8	-	2996	3113.37	58.76	-	-	-	1045	1195.87	58.11	-	-	-	467	533.23	40.01	-	-
abz9	-	2947	3123.93	86.83	-	-	-	898	979.40	54.16	-	-	-	245	310.17	47.33	-	-
la21	3560	3726	4001.03	206.00	4.66	12.39	1570	1675	1801.07	51.35	6.69	14.72	868	913	960.70	38.28	5.18	10.68
la24	3553	3692	3835.33	86.74	3.91	7.95	1570	1570	1647.43	81.40	0.00	4.93	693	752	758.73	6.95	8.51	9.49
la25	3313	3332	3550.00	112.87	0.57	7.15	1430	1430	1490.00	78.64	0.00	4.20	874	890	941.63	52.13	1.83	7.74
la27	9090	9512	10253.53	372.05	4.64	12.80	5757	**5653**	6426.50	278.47	-1.81	11.63	4203	4364	4700.83	173.31	3.83	11.84
la29	8744	9730	10125.77	223.61	11.28	15.80	5959	**5864**	6530.07	234.10	-1.59	9.58	4392	4665	5054.07	212.58	6.22	15.07
la38	2122	2206	2383.97	98.70	3.96	12.35	401	**344**	486.67	45.13	-14.21	21.36	0	0	1.30	5.66	0.00	-
la40	2078	2078	2197.27	63.66	0.00	5.74	51	76	83.70	11.80	49.02	64.12	0	0	0.00	0.00	0.00	0.00
					3.63	10.57					4.76	17.15					3.82	11.42

Tabla 5.3: Resultados del JSP minimizando el TWT con tiempos de procesamiento difusos.

Instancia	f=1.3			f=1.5			f=1.6		
	TWT			TWT			TWT		
	Mejor	Medio	SD	Mejor	Medio	SD	Mejor	Medio	SD
ft10	1458.75	1650.22	148.30	461.25	523.39	52.05	243.25	245.83	2.54
ft20	13460.00	14011.32	753.57	11519.50	12156.46	682.20	10559.50	11067.81	564.71
abz7	3469.00	3705.70	136.10	1444.00	1585.03	110.46	774.50	880.76	86.39
abz8	3117.75	3254.37	155.82	1155.50	1301.27	70.63	581.50	684.43	82.69
abz9	3147.75	3282.42	62.24	1042.00	1121.08	76.21	460.00	478.47	18.11
la21	3944.25	4210.70	196.91	1839.25	2037.98	89.92	1102.00	1193.78	55.95
la24	3780.75	4011.17	141.71	1675.00	1768.24	103.57	940.00	970.29	30.99
la25	3487.00	3769.13	196.93	1591.25	1704.14	92.82	1036.50	1138.78	54.03
la27	9837.75	10600.26	324.61	6170.75	6670.31	254.86	4571.50	4953.95	212.86
la29	10028.20	10375.61	270.02	6363.50	6900.41	409.03	4561.25	5294.26	256.42
la38	2451.75	2622.89	96.66	583.50	671.76	38.64	107.25	137.35	30.73
la40	2387.00	2480.69	48.33	284.50	299.18	16.56	24.25	30.94	6.87

CAPÍTULO 6

Conclusiones y trabajo futuro

En este capítulo se explicarán las conclusiones extraídas del trabajo realizado y se propondrán posibles líneas para continuar con la investigación.

6.1 Conclusiones

En este trabajo se ha diseñado e implementado un algoritmo memético que integra una componente evolutiva con un algoritmo de búsqueda monoestado. Obtiene así un equilibrio entre diversificación e intensificación, dando lugar a un método de búsqueda aplicable a diferentes variantes del JSP que resulta competitivo con las mejores técnicas del momento, tal y como se demuestra en el Capítulo 5.

No solo se ha dado cobertura a problemas que habían recibido atención previamente en la literatura y con los que se puede realizar una comparación directa, sino que incluso se ha solucionado una variante del problema que no había sido considerada con anterioridad, el JSP minimizando el TWT con tiempos de procesamiento difusos. De esta manera este trabajo se convierte en una primera referencia y los resultados aquí expuestos servirán como comparación en trabajos posteriores.

Las técnicas propuestas, aunque no totalmente innovadoras, pues están sustentadas en trabajos previos altamente reconocidos, han demostrado un gran rendimiento en este caso concreto y podrían ser fácilmente extendidas a otros problemas de planificación temporal de tareas.

Además, se ha evaluado el funcionamiento de las dos componentes del algoritmo memético por separado (Apéndice A), para corroborar que realmente existe una sinergia entre ambas metaheurísticas.

Para terminar, decir que se han cumplido todos los objetivos expuestos en el Capítulo 1. En un trabajo planteado como una iniciación a la investigación, se han estudiado, a partir de diversas fuentes, el problema a resolver y las técnicas de Inteligencia Artificial adecuadas para resolverlo. Se han generalizado algunos resultados teóricos ya existentes para poder explotar el conocimiento heurístico a la hora de diseñar e implementar un método de resolución, obteniendo unos resultados que sitúan a este trabajo en el estado del arte de las soluciones

metaheurísticas para resolver el *job shop scheduling problem*.

6.2 Trabajo futuro

Pese a los buenos resultados obtenidos, todavía quedan muchos aspectos que podrían ser estudiados para intentar mejorar aún más el rendimiento de las metaheurísticas propuestas.

El rendimiento puede ser medido de dos maneras distintas, en función de cuál sea el aspecto que más interese mejorar ahora que ya se conocen los resultados de una primera versión. Por un lado, se puede intentar reducir el tiempo de ejecución (Sección 6.2.1), haciendo los algoritmos más ligeros, tratando de no empeorar la calidad de las soluciones en el proceso. Por otro lado se puede intentar mejorar la calidad de las soluciones obtenidas (Sección 6.2.2), añadiendo para ello más componentes a los algoritmos que, explotando más conocimiento del problema, sean capaces de dirigir la búsqueda de forma más efectiva.

6.2.1 Reducción del tiempo de ejecución

La forma de reducir el tiempo de ejecución es detectar aquellas áreas en los algoritmos en las que se realiza trabajo que no contribuye a la mejora de la calidad de las soluciones.

En el JSP uno de los puntos a los que históricamente se ha dedicado más atención para agilizar el trabajo de búsqueda es el uso de estimaciones a la hora de evaluar las soluciones. Evaluar la calidad de una solución no es un proceso especialmente costoso en sí, su complejidad para las funciones objetivo consideradas es $\mathcal{O}(n \cdot m)$ donde n es el número de trabajos y m el número de máquinas. El problema es que durante el proceso de búsqueda se evalúa una gran cantidad de soluciones, sobre todo cuando se utilizan funciones de vecindad muy grandes. La función de estimación puede utilizarse como sustituto de la función de evaluación exacta o como filtro, para obtener una idea preliminar de cuál es su calidad antes de decidir si merece la pena realizar la evaluación completa o no. Una cualidad deseada en las estimaciones es que sean una cota inferior de la calidad real, es decir, que su valor no sea más grande del que en realidad es.

La idoneidad del uso de estimaciones ya ha sido evaluada superficialmente durante la realización de este trabajo, aunque no se ha comentado durante el texto principal. Las conclusiones obtenidas fueron que en el caso del querer minimizar el *makespan* existe una función de estimación [35] que, con una complejidad $\mathcal{O}(1)$, permite obtener resultados muy buenos, equiparándose la calidad de las soluciones obtenidas y usando una cantidad de tiempo muy inferior. Sin embargo, su extensión para trabajar con otras funciones objetivo, concretamente para el TWT, no es tan buena, ya que su complejidad asciende a $\mathcal{O}(n)$ pero además requiere de realizar acciones adicionales durante el proceso de evaluación que provocan que su complejidad aumente hasta $\mathcal{O}(n^2 \cdot m)$, con lo que al final acaba siendo más lenta, como también se concluye en [2]. Sumado a esto, la estimación que se obtiene es realmente mala pues su valor está muy lejos del que realmente debería ser. El objetivo sería encontrar una función de estimación que mejore en ambos aspectos, tanto en su coste computacional como en la calidad de la misma.

Otra posible optimización, que afectaría únicamente a la función objetivo para el TWT, sería considerar únicamente un camino crítico, en lugar de uno por trabajo. Este camino crítico se seleccionaría como el que más contribuyese a la medida. Sin embargo, podría dar problemas si hay un camino que, aun siendo lo más corto posible, sigue siendo más largo que el resto pues siempre se tomaría en consideración solo ese y no se optimizaría el resto. Por ello, quizá tenga más sentido una selección aleatoria asignando probabilidades dependientes de la magnitud del valor; de esta manera, se darían posibilidades a todos los caminos y se solucionaría la situación antes descrita.

6.2.2 Mejora de la calidad de las soluciones

Para mejorar la calidad de las soluciones es necesario añadir componentes adicionales a los algoritmos de búsqueda que aprovechen aun más el conocimiento del problema disponible.

El algoritmo evolutivo ya utiliza la codificación más ampliamente aceptada y, por su naturaleza, presenta pocas posibilidades de mejora en este aspecto. Sin embargo, relacionado con la codificación están los operadores de cruce. En este trabajo se ha considerado un único operador de cruce, sin embargo, existen otras muchas alternativas que podrían ser más eficaces extrayendo las características interesantes de los padres o combinando los genes de estos para dar lugar a los descendientes [4]. En cuanto al resto de componentes del algoritmo evolutivo, ya se han probado los más habituales, sin embargo, podrían probarse algoritmos de selección alternativos para mover la presión selectiva a esta fase, que es más frecuente que durante el reemplazo, tal y como se ha hecho.

En cuanto a la búsqueda tabú implementada, podría ser extendida para incluir memoria a más largo plazo, con un mecanismo de detección de ciclos, que aunque fue inicialmente identificado como innecesario, pues en el proceso de hibridación las ejecuciones de la búsqueda monoestado iban a ser muchas pero cortas, podría ayudar en determinadas situaciones.

También podrían diseñarse nuevas funciones de vecindad o probarse otras ya propuestas en la literatura [22], con el objetivo de permitir a la búsqueda monoestado moverse a otras soluciones más interesantes. Sin embargo, se debería intentar no hacer esta mejora en las funciones de vecindad, a costa de aumentar su tamaño pues esto supondría un aumento excesivo del tiempo de ejecución que podría no compensar la mejora obtenida.

Por último, podría incluirse una mejora que beneficiaría a ambas estrategias y que consiste en el uso de heurísticas constructivas (*heuristic seeding*) para la generación de alguna solución de la población inicial. De esta manera habría un buen punto de partida para la búsqueda en lugar de uno totalmente aleatorio.

6.3 Competencias del grado

En línea con la competencia específica del grado CE25, este trabajo, para el que ha sido necesaria una comprensión global de la disciplina, ha consistido en un proyecto original en el ámbito de las tecnologías específicas de la Ingeniería en Informática, en particular, sintetizando e integrando competencias de la Tecnología Específica de Computación. De entre estas cabe

destacar como especialmente relevantes las competencias específicas CM3 (Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.) y CM4 (Capacidad para conocer los fundamentos, paradigmas y técnicas propias de los sistemas inteligentes y analizar, diseñar y construir sistemas, servicios y aplicaciones informáticas que utilicen dichas técnicas en cualquier ámbito de aplicación), que profundizan en la consecución de la competencia común CO15 (Conocimiento y aplicación de los principios fundamentales y técnicas básicas de los sistemas inteligentes y su aplicación práctica.).

Bibliografía

[1] D. Applegate y W. Cook. "A Computational Study of the Job-Shop Scheduling Problem". En: *ORSA Journal on Computing* 3.2 (1991), págs. 149-156. DOI: 10.1287/ijoc.3.2.149.

[2] C. Bierwirth y J. Kuhpfahl. "Extended GRASP for the job shop scheduling problem with total weighted tardiness objective". En: *European Journal of Operational Research* 261.3 (2017), págs. 835-848. DOI: 10.1016/j.ejor.2017.03.030.

[3] C. Bierwirth. "A generalized permutation approach to job shop scheduling with genetic algorithms". En: *OR Spektrum* 17.2-3 (1995), págs. 87-92. DOI: 10.1007/bf01719250.

[4] C. Bierwirth, D. C. Mattfeld y H. Kopfer. "On permutation representations for scheduling problems". En: *Parallel Problem Solving from Nature PPSN IV* (1996), págs. 310-318. DOI: 10.1007/3-540-61723-x_995.

[5] G. Bortolan y R. Degani. "A review of some methods for ranking fuzzy subsets". En: *Readings in Fuzzy Sets for Intelligent Systems* (1993), págs. 149-158. DOI: 10.1016/b978-1-4832-1450-4.50016-x.

[6] M. Brunelli y J. Mezei. "How different are ranking methods for fuzzy numbers? A numerical study". En: *International Journal of Approximate Reasoning* 54.5 (2013), págs. 627-639. DOI: 10.1016/j.ijar.2013.01.009.

[7] K. Bülbül. "A hybrid shifting bottleneck-tabu search heuristic for the job shop total weighted tardiness problem". En: *Computers & Operations Research* 38.6 (2011), págs. 967-983. DOI: 10.1016/j.cor.2010.09.015.

[8] M. Dell'Amico y M. Trubian. "Applying tabu search to the job-shop scheduling problem". En: *Annals of Operations Research* 41.3 (1993), págs. 231-252. DOI: 10.1007/bf02023076.

[9] Dimitri van Heesch. *Doxygen*. 27 de dic. de 2018. URL: http://www.doxygen.nl/.

[10] D. Dubois y H. Prade. "Fuzzy numbers: An overview". En: *Readings in Fuzzy Sets for Intelligent Systems* (1993), págs. 112-148. DOI: 10.1016/b978-1-4832-1450-4.50015-8.

[11] P. Fortemps. "Jobshop scheduling with imprecise durations: a fuzzy approach". En: *IEEE Transactions on Fuzzy Systems* 5.4 (1997), págs. 557-569. DOI: 10.1109/91.649907.

[12] S. French. *Sequencing and scheduling: an introduction to the mathematics of the job-shop*. Ellis Horwood series in mathematics and its applications. Ellis Horwood, 1982.

[13] M. R. Garey, D. S. Johnson y R. Sethi. "The Complexity of Flowshop and Jobshop Scheduling". En: *Mathematics of Operations Research* 1.2 (1976), págs. 117-129. DOI: 10.1287/moor.1.2.117.

[14] B. Giffler y G. L. Thompson. "Algorithms for Solving Production-Scheduling Problems". En: *Operations Research* 8.4 (1960), págs. 487-503. DOI: 10.1287/opre.8.4.487.

[15] F. Glover. "Future paths for integer programming and links to artificial intelligence". En: *Computers & Operations Research* 13.5 (1986), págs. 533-549. DOI: 10.1016/0305-0548(86)90048-1.

[16] I. Gonzalez-Rodríguez, C. R. Vela y J. Puente. "A Memetic Approach to Fuzzy Job Shop Based on Expectation Model". En: *2007 IEEE International Fuzzy Systems Conference* (2007). DOI: 10.1109/fuzzy.2007.4295450.

[17] I. González-Rodríguez, C. R. Vela, J. Puente y R. Varela. "A New Local Search for the Job Shop Problem with Uncertain Durations". En: *Proceedings of the Eighteenth International Conference on International Conference on Automated Planning and Scheduling*. ICAPS'08 (2008), págs. 124-131.

[18] S. Heilpern. "The expected value of a fuzzy number". En: *Fuzzy Sets and Systems* 47.1 (1992), págs. 81-86. DOI: 10.1016/0165-0114(92)90062-9.

[19] ISO. *ISO/IEC 19217:2015 Information technology — Programming languages — C++ Extensions for concepts*. First edition. 2015.

[20] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. Fifth edition. 2017.

[21] P. Jorge, V. C. R. e I. González-Rodríguez. "Fast Local Search for Fuzzy Job Shop Scheduling". En: *Frontiers in Artificial Intelligence and Applications* 215.ECAI 2010 (2010), págs. 739744. DOI: 10.3233/978-1-60750-606-5-739.

[22] J. Kuhpfahl y C. Bierwirth. "A study on local search neighborhoods for the job shop scheduling problem with total weighted tardiness objective". En: *Computers & Operations Research* 66 (2016), págs. 44-57. DOI: 10.1016/j.cor.2015.07.011.

[23] P. J. M. van Laarhoven, E. H. L. Aarts y J. K. Lenstra. "Job Shop Scheduling by Simulated Annealing". En: *Operations Research* 40.1 (1992), págs. 113-125. DOI: 10.1287/opre.40.1.113.

[24] D. Lei. "Fuzzy job shop scheduling problem with availability constraints". En: *Computers & Industrial Engineering* 58.4 (2010), págs. 610-617. DOI: 10.1016/j.cie.2010.01.002.

[25] J. Lenstra, A. R. Kan y P. Brucker. "Complexity of Machine Scheduling Problems". En: *Studies in Integer Programming* (1977), págs. 343-362. DOI: 10.1016/s0167-5060(08)70743-x.

[26] O. Management Group. *OMG Unified Modeling Language – Version 2.5.1*. Dic. de 2017.

[27] D. Mattfeld. *Evolutionary Search and the Job Shop: Investigations on Genetic Algorithms for Production Scheduling*. Production and Logistics. Physica-Verlag, 1996.

[28] J. Muth y G. Thompson. *Industrial Scheduling*. International series in management. Prentice-Hall, 1963.

[29] E. Nowicki y C. Smutnicki. "A Fast Taboo Search Algorithm for the Job Shop Problem". En: *Management Science* 42.6 (1996), págs. 797-813. DOI: 10.1287/mnsc.42.6.797.

[30] J. J. Palacios, C. R. Vela, I. González-Rodríguez y J. Puente. "Schedule Generation Schemes for Job Shop Problems with Fuzziness". En: *Frontiers in Artificial Intelligence and Applications* 263.ECAI 2014 (2014), págs. 687-692. DOI: 10.3233/978-1-61499-419-0-687.

[31] J. J. Palacios, J. Puente, C. R. Vela e I. González-Rodríguez. "Benchmarks for fuzzy job shop problems". En: *Information Sciences* 329 (2016), págs. 736-752. DOI: 10.1016/j.ins.2015.09.042.

[32] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. 3rd. Springer Publishing Company, Incorporated, 2008.

[33] I. Sommerville. *Software Engineering*. 10th edition. Pearson, 2015.

[34] B. Stroustrup. *The C++ Programming Language*. 4th edition. Addison-Wesley Professional, 2013.

[35] É. D. Taillard. "Parallel Taboo Search Techniques for the Job Shop Scheduling Problem". En: *ORSA Journal on Computing* 6.2 (1994), págs. 108-117. DOI: 10.1287/ijoc.6.2.108.

[36] E.-G. Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.

[37] L. Zadeh. "Fuzzy sets". En: *Information and Control* 8.3 (1965), págs. 338-353. DOI: 10.1016/s0019-9958(65)90241-x.

Estudio de sinergia

En este apéndice se complementan los resultados experimentales del Capítulo 5. Para ello se analizará cómo la componente evolutiva y la componente de búsqueda monoestado del algoritmo memético colaboran juntas para encontrar soluciones que son mejores que la que encontrarían ambas técnicas por separado. Aunque es algo que se ha dado por supuesto durante el trabajo, pues los algoritmos meméticos son algoritmos bien establecidos, es posible hacer la comprobación de forma experimental.

Para llevar a cabo el experimento se ejecutarán ambas componentes por separado durante un tiempo aproximadamente igual al tiempo invertido por el algoritmo memético. La forma de determinar el tiempo que asignar no es trivial, pues entre distintas ejecuciones del algoritmo memético existían diferencias, ya que se utilizó un criterio de parada dinámico y no por tiempo. Tomar la media de los tiempos de ejecución y parar cuando se alcance este valor no parece lo más justo pues podría detenerse la búsqueda durante una fase de mejora y por otro lado podría alargarse innecesariamente una ejecución poco prometedora. Es por ello que simplemente se van a establecer parámetros de configuración que den tiempos de ejecución del mismo orden que los obtenidos para el memético, de tal manera que cada ejecución individual tenga independencia para alargarse o acortarse en función de lo prometedora que sea. Aunque no es la forma más precisa, sí que es la que ofrece unos resultados más justos.

Aunque es posible realizar el estudio de sinergia para todas las funciones objetivo y todas las instancias consideradas, no es realmente necesario. Si se constata que para una función objetivo ambas componentes son capaces de trabajar juntas, combinando sus puntos fuertes, los resultados pueden ser extrapolados al resto. Al mismo razonamiento es aplicable para las distintas instancias.

Dicho esto, las instancias a considerar serán el `ft10` y `ft20` de Fisher y Thompson [28]. La elección de estas instancias no es casual, siendo dos de las más conocidas y con más historia dentro de este ámbito. La función objetivo a considerar será el TWT con tiempos de procesamiento difusos. Se utilizarán los resultados para el algoritmo memético presentados en la Tabla 5.3.

Para realizar el análisis de la búsqueda tabú se generarán soluciones aleatorias y se utilizarán como punto inicial para el algoritmo configurado de la misma manera que cuando se utilizó en el algoritmo memético. En este punto, entra la discusión de si conviene permitir a la búsqueda

tabú ejecutarse durante más iteraciones o generar más soluciones de inicio. En este caso, es mejor decantarse por lo segundo porque la búsqueda tabú implementada no tiene un mecanismo de memoria a largo plazo para poder detectar ciclos y, por tanto, si se le permiten demasiadas iteraciones, podría entrar en un bucle, realizando trabajo que no va a llevar a una solución mejor. No obstante, si la búsqueda tabú fuese más sofisticada, utilizar ejecuciones más largas posiblemente diera un mejor rendimiento.

La configuración del algoritmo evolutivo es más sencilla, basta utilizar los mismos parámetros que los utilizados para el algoritmo memético, aumentando el tamaño de la población para que los tiempos sean del mismo orden.

Los resultados obtenidos se pueden ver en la Tabla A.1. Aunque al igual que en el Capítulo 5 se han realizado 30 ejecuciones, se presentan únicamente los mejores resultados encontrados. La comparación se realizará calculando el error relativo con respecto a la solución obtenida por el algoritmo memético, que sirve de referencia. Ninguna de las dos componentes es capaz de alcanzar los resultados obtenidos por el algoritmo memético. La componente evolutiva, aunque se queda relativamente cerca, incluso igualando al memético en algunas instancias, no está a la misma altura pues el error relativo es alto en muchos otros casos. Por otro lado, la búsqueda tabú obtiene resultados mucho peores, muy lejos de los del memético y también de los del evolutivo.

Ambos resultados tienen explicación. Por un lado, la búsqueda tabú es un algoritmo de búsqueda monoestado que se basa en la modificación de una solución inicial, a través de las funciones de vecindad, para obtener soluciones cada vez mejores. El problema que surge es que el espacio de soluciones es tan sumamente grande que para llegar a las soluciones buenas hay que partir de una bastante cercana. La razón es que, aunque la búsqueda tabú es capaz de escapar de los óptimos locales más pequeños, no es capaz de escapar de aquellos de tamaño medio. Es por ello que comúnmente los algoritmos de búsqueda monoestado, en lugar de recibir una solución inicial aleatoria, reciben una solución relativamente buena, generada con un algoritmo sencillo, como un algoritmo voraz. Sin embargo, para este caso no se desarrolló ningún tipo de mecanismo de generación de soluciones, porque el objetivo era la hibridación, y no tendría uso alguno en este contexto. En conclusión, la búsqueda tabú es un algoritmo capaz de intensificar la búsqueda en una zona concreta del espacio de soluciones pero no de diversificar.

Por otro lado el algoritmo evolutivo es capaz de explorar zonas mucho más grandes del espacio de soluciones. Sin embargo, no es capaz de mejorar al máximo las soluciones que encuentra, por lo que es posible que, aun estando cerca de un óptimo, tarde mucho en alcanzarlo. Su convergencia es por tanto más lenta, y podría incluso ocurrir que ignore completamente un óptimo para ir a explorar a otra zona, debido a que tiende a diversificar la búsqueda. Esto no quiere decir que los algoritmos evolutivos no sean capaces de encontrar soluciones óptimas, sino que aun partiendo de una buena población inicial, suelen tardar más tiempo en hacerlo. Prueba de lo anterior es que solo la componente evolutiva ha sido capaz de igualar al algoritmo memético en algún caso, cuyos resultados no está garantizado que sean óptimos, pero deberían estar relativamente cerca.

Descritas ambas situaciones anteriores, es fácil entender los buenos resultados obtenidos

por el algoritmo memético. Las soluciones que va explorando la componente evolutiva están suficientemente cerca de soluciones óptimas (ya sea globales o locales) y al utilizarlas como punto de entrada para la búsqueda tabú, esta es capaz de llevarlas hasta estos valores óptimos. Además, como es capaz de escapar de los óptimos locales más pequeños, es capaz de llegar a las mejores soluciones de la zona del espacio de soluciones desde el que fue iniciada. La componente evolutiva, por otro lado, extraerá los genes mejorados y los recombinará para llevar la búsqueda a zonas aún mejores. De esta manera, ambas componentes cooperan, uniendo sus puntos fuertes y contrarrestando los débiles.

Antes de terminar, conviene señalar que aunque la componente monoestado y la evolutiva por separado se quedan lejos del memético, no se debe a que sean malas metaheurísticas, únicamente que no están tan optimizadas como estarían si hubiesen sido diseñadas desde un principio para trabajar por separado. A la hora de hibridar dos algoritmos, no suele ser conveniente que estén excesivamente ajustados, pues podría provocarse una convergencia prematura.

Con este análisis se ha comprobado la hipótesis de que entre la componente evolutiva y la monoestado del algoritmo memético existe una sinergia que permite alcanzar soluciones mejores que las que se obtendrían de forma independiente.

Tabla A.1: Resultados del estudio de sinergia.

Instancia	f=1.3					f=1.5					f=1.6				
	Alg. memético	Algoritmo evolutivo		Búsqueda tabú		Alg. memético	Algoritmo evolutivo		Búsqueda tabú		Alg. memético	Algoritmo evolutivo		Búsqueda tabú	
	TWT	TWT	RE (%)	TWT	RE (%)	TWT	TWT	RE (%)	TWT	RE (%)	TWT	TWT	RE (%)	TWT	RE (%)
ft10	1458.75	1699.00	16.47	2019.00	38.41	461.25	576.50	24.99	812.00	76.04	243.25	243.25	0.00	328.50	35.05
ft20	13460.00	13509.20	0.37	17852.50	32.63	11519.50	11648.20	1.12	16899.20	46.70	10559.50	11109.80	5.21	15838.20	49.99