Universidad de Valladolid

Facultad de Ciencias

TRABAJO FIN DE GRADO

Grado en Estadística

PROGRAMACIÓN DE TAREAS: Interval Scheduling

Autor:
Rocío Íñigo Lobo

Tutores:
Jesús A. Tapia García
Jesús Sáez Aguado

Agradecimientos

Este trabajo no habría podido llevarse a cabo si no fuera gracias a las personas que me han apoyado durante su desarrollo, así como durante el transcurso de la carrera universitaria. En especial:

Mis compañeros, que con el tiempo han pasado a ser buenos amigos, por su apoyo en aquellos momentos de duda o frustración.

A mi pareja, por apoyarme durante el desarrollo de este trabajo y estar a mi lado cuando lo he necesitado.

A los profesores, en especial a mis tutores Jesús Sáez y Jesús Tapia, por aportarme sus conocimientos y guiarme durante el desarrollo de este trabajo.

Y por último, a mis padres, por su apoyo durante estos años y por su paciencia conmigo en aquellas situaciones en que me sobrepasaba el estrés.

Resumen

El objetivo de este Trabajo de Fin de Grado es realizar una revisión de algunas de las variantes más importantes o conocidas de los problemas de *Interval Scheduling* y otros problemas relacionados.

En concreto, para cada problema, se establece una descripción del mismo y se explican los métodos más notables para encontrar una solución.

La descripción de cada problema consta de los datos de entrada con los que se cuenta inicialmente y del objetivo que se pretende cumplir una vez obtenida la solución del mismo. Además de esto, en aquellos casos en los que se considera necesario, se establece la formulación de programación entera correspondiente.

Para algunos problemas, junto con una descripción de los diferentes algoritmos utilizados para obtener soluciones y sus demostraciones, se establece la implementación en *Xpress* realizada en base a estas; así como las soluciones obtenidas para diferentes ficheros de datos con un formato acorde a la correspondiente variante del problema.

Abstract

The aim of this Final Degree Project is to go over thoroughly all the most known and important variants of the *Interval Scheduling* problems and other related problems.

Specifically, for each problem, it can be found a description and also an explanation of the most relevant methods used to find out a solution.

The description of each problem consists on the input data that is known in the beginning and also the goal that should be reached when a solution is obtained. In addition, it has been added the integer programming formulation for those cases when it is considered necessary.

For some problems, along with the description of the different algorithms used to get solutions and their proofs of correctness, it is explained the implementation in *Xpress* based on their explanation. Some results are also shown for the different data files with the appropiate format for the corresponding problem.

Índice general

Índice de figuras

Introducción.

El **objetivo principal** de este trabajo es realizar una revisión teórica sobre los diferentes problemas más conocidos de *Interval Scheduling* y otros relacionados. Además de esto, se pretende comprobar si realmente merece la pena el uso de algoritmos que realicen una búsqueda exhaustiva del óptimo o si con aplicar métodos como el algoritmo *greedy*, heurísticas de mejora o metaheurísticas, es suficiente.

En el **Capítulo 1** se da una descripción de los problemas conocidos como *Interval Scheduling Problems*, que se caracterizan porque las tareas presentan tiempos de inicio fijos.
Se describen a continuación los diferentes datos de entrada que se pueden encontrar para este tipo de problemas, junto con la nomenclatura que se va a usar a lo largo de la memoria para referirse a los mismos.
Una vez introducido lo que es un problema de *Interval Scheduling* se explican de forma general las diferentes variantes que se van a tratar en las secciones de esta memoria y a continuación se da una visión más detallada de cada una de ellas.
Además de esto, se explican las diferentes aplicaciones que se pueden dar a este tipo de problemas.

En el **Capítulo 2** se tratan los problemas que implican el uso de una sola máquina o procesador: el *Interval Scheduling Maximization Problem*, su versión con prioridades, el *Weighted Interval Scheduling Problem*, los problemas con *deadlines*, *Interval Scheduling with Deadlines and Profits*, *Interval Scheduling with Deadlines and Release Times* y los problemas conocidos como *Discrete Interval Scheduling Problem*, *Minimize Maximum Lateness* y *Minimize Mean End Time*.

Los dos primeros son los problemas básicos de *Interval Scheduling*: el primero, tratado en la **Sección 2.1** busca maximizar el número de tareas a ejecutar en un procesador y el segundo, en la **Sección 2.2**, busca maximizar el beneficio total generado por estas tareas.
Para el primero de estos problemas, se explica el algoritmo *greedy*, método que permite obtener el óptimo y se prueba que esto realmente es así. Se indica junto con esto la complejidad del algoritmo.
Para el segundo problema se explican varios métodos para obtener soluciones, aun-

que en este caso no garanticen el óptimo.
Tanto para el primero como para el segundo problema, se resuelven los problemas de programación entera con *Xpress* y el algritmo *greedy* para varios ficheros, y además para el segundo se aplican dos versiones de la heurística de mejora basada en búsqueda local.
Para el primer problema se comprueba que, efectivamente, tanto el *solver* de *Xpress* como el algoritmo *greedy* devuelven una solución óptima.

El resto de problemas no se ajustan de forma tan rigurosa a la definición de problema de *Interval Scheduling*, ya que sus tiempos de inicio no son fijos, sino que forman parte de la solución que se desea obtener.
Así, el problema de *Interval Scheduling with Deadlines and Profits*, tratado en la **Sección 2.3**, tiene como objetivo maximizar los beneficios ejecutando las tareas que lo permitan antes de sus correspondientes *deadlines*.
Para este caso, se explica el algoritmo *greedy* junto con su complejidad y se implementa el problema de programación entera y el algoritmo *greedy*. Aunque no se tiene garantía de que el algoritmo *greedy* ofrezca una solución óptima, se comprueba que los valores obtenidos son mejores o iguales que los que proporciona el *solver* de *Xpress* en un tiempo razonable.
Por su parte, el problema de *Interval Scheduling with Deadlines and Release times*, en la **Sección 2.4** busca ejecutar el número máximo de tareas antes de sus *deadlines* y a partir de un momento determinado para cada tarea, conocido como r_j o *release time*. En este caso, se explican los métodos para buscar soluciones, pero no se implementa ninguno de ellos.
En la **Sección 2.5** se presenta de forma genérica el caso del *Discrete Interval Scheduling Problem*.
El problema de *Minimize Maxmimum Lateness* se trata en la **Sección 2.6** y presenta el algoritmo *greedy* que en este caso sí encuentra el óptimo, como se prueba después junto con su complejidad. Se implementa tanto este algoritmo como el problema de programación entera, comprobándose que, efectivamente, dan los mismos resultados o en todo caso el algoritmo *greedy* obtiene soluciones mejores.
En el caso del *Minimize Mean End Time Problem*, tratado en la **Sección 2.7**, se implementa únicamente el algoritmo *greedy* que se sabe, obtiene soluciones óptimas.

En el **Capítulo 3** se tratan los problemas que implican el uso de varias máquinas o procesadores, bien con un número fijo de máquinas o bien con un número variable.

En primer lugar, se introducen en las **Secciones 3.1.1 y 3.1.2** los casos sin y con prioridades en los que se busca maximizar el número de tareas y beneficios respectivamente, con más de un procesador. En el caso con prioridades se introducen además diferentes variantes del problema: *Interval Scheduling* con disponibilidades de máquinas, *Interval Scheduling* con tareas requeridas e *Interval Scheduling* jerárquico.

Después se trata el problema de *Interval Scheduling Maximizing Minimum Coverage*, en la **Sección 3.2** cuyo objetivo es seleccionar un subconjunto de tareas para ser ejecutadas por un número dado de máquinas de forma que el mínimo de máquinas que estén ocupadas en un tiempo dado, sea lo mayor posible. Se explican con este los métodos utilizados para buscar soluciones.

Por último, en la **Sección 3.3** se explica el problema conocido como *Interval Partitioning*, el caso en que el número de máquinas es variable y el objetivo es minimizar el número de máquinas utilizado para ejecutar todas las tareas. En este caso, se explica el método para obtener el óptimo y se implementa tanto este como el problema de programación entera. Se observa que efectivamente el óptimo y la solución del algoritmo *greedy* coinciden.

Finalmente, se establecen las conclusiones extraídas tras la realización del trabajo.

Capítulo 1

Problemas de programación de tareas con intervalos.

1.1. Descripción

Los problemas de programación de tareas con intervalos o *Interval Scheduling* son problemas de programación de tareas en los que el tiempo de inicio es fijo y se conocen también como *fixed job scheduling problems* o *k-track assignment problems* [**4**].

Su historia se remonta a los años 50, cuando Dantzig y Fulkerson describieron un problema de *scheduling* para un petrolero. A su vez, Fulkerson y Ford resolvieron un problema de *Interval Scheduling* usando el teorema de Dilworth. Estos problemas parten de la siguiente información, [**5**][**6**], para n tareas dadas:

- El **tiempo de inicio** de la tarea j-ésima: s_j
- El **tiempo de finalización** de la tarea j-ésima: f_j
- En algunos casos, la **prioridad** de la tarea j-ésima: p_j
- En otros casos, el **coste** de la tarea j-ésima: c_j o el **beneficio** de la tarea j-ésima: b_j
- En algunos problemas, un **tiempo límite o *deadline***: d_j, cuyo valor no deberá superar el de f_j

A partir de este punto, se utilizará **tiempo de procesamiento**, t_j, para hacer referencia al tiempo comprendido entre el tiempo de inicio, s_j, y el tiempo de finalización, f_j, de una misma tarea j, con j = 1,...,n. Es decir, $t_j = f_j$ - s_j. Además, el intervalo $[s_j,\ f_j)$ se conocerá como **intervalo de procesamiento**. Véase que

el intervalo es abierto en el extremo correspondiente al tiempo de finalización, de forma que una tarea cuyo tiempo de inicio sea igual a f_j no se solaparía con esta. La longitud de este intervalo será igual al tiempo de procesamiento, t_j, que será siempre positivo, es decir $s_j < f_j$.

Un conjunto de tareas debe ser **compatible**, [**5**][**6**], basándose en unas u otras propiedades de las mismas.
Un ejemplo de esto se da cuando el problema exige que la ejecución de dos tareas en una misma máquina esté condicionada a que sus intervalos de procesamiento no se solapen, como se explicará formalmente más adelante.
Los subconjuntos de tareas obtenidos como solución y su correspondiente asignación, en caso de darse, a las máquinas de las que se disponga, se conoce como ***schedule*** o **planificación**.

Dada esta información, pueden presentarse diferentes objetivos o variantes del problema, teniendo en cuenta que cada máquina puede procesar como mucho una tarea al mismo tiempo y que siempre está disponible en caso de no estar ejecutando ninguna, [**7**]. Además, las tareas solo serán asignadas a una máquina en caso de haber más de una.

Si solo se dispone de **una máquina**, pueden darse las siguientes situaciones, dependiendo de los datos de los que se disponga:

- Si se desconoce la prioridad de cada tarea, es suficiente con buscar el número máximo de tareas que puedan realizarse sin que estas se solapen entre sí. Este problema se conoce como ***Interval Scheduling Maximization Problem*** y se trata en el **Capítulo 2.1.**
- En caso de conocer la prioridad de cada tarea, lo óptimo sería seleccionar un subconjunto de tareas que maximice la prioridad total. Este problema es conocido como ***Weighted Interval Scheduling Problem*** y se trata en el **Capítulo 2.2**.
- En los capítulos 2.3 y 2.4 se tratan dos variantes del problema en las que se trabaja con tiempos límite para la ejecución de cada tarea o *deadlines* conocidos. Este problema se conoce como ***Interval Scheduling with Deadline*** con la diferencia de que en el primer caso, **Capítulo 2.3**, se conoce el beneficio, b_j que genera cada tarea y el objetivo es maximizar el beneficio generado por el subconjunto de tareas elegido; mientras que en el segundo caso se conoce el valor de los tiempos de salida o *release times* de cada tarea, r_j, es decir, el momento a partir del cual puede ser ejecutada dicha tarea, y el objetivo es ejecutar el número máximo de tareas posible antes de sus correspondientes *deadlines*, como se explica en el **Capítulo 2.4**.
- Otra variante del problema de *scheduling* con restricciones de tiempo es la conocida como ***Discrete Interval Scheduling Problem***, donde se puede

seleccionar el valor de s_j de un conjunto de valores diferentes y el objetivo es, de nuevo, maximizar el número de tareas a ejecutar. Una introducción a esta variante se encuentra en el **Capítulo 2.5**.

- En el **Capítulo 2.6** se trata el problema ***Minimize Maximum Lateness*** que, partiendo de un conjunto de tareas cuya duración t_j es conocida, así como el valor de su *deadline*, tiene como objetivo obtener una planificación de las tareas que minimice el retraso máximo o *maximum lateness*.

- En una última variante, problema que se conocerá como ***Minimize Mean End Time***, el objetivo es minimizar el tiempo medio de finalización conocidos el tiempo de procesamiento y la prioridad de cada tarea como se explica en el **Capítulo 2.7.**

En caso de contar con **más de una máquina** para ejecutar las tareas, se describen las siguientes posibilidades:

- En primer lugar, se describen los casos correspondientes a los problemas conocidos como *Interval Scheduling Maximization Problem* y *Weighted Interval Scheduling Problem*, para el caso en el que puede contarse con más de una máquina y el **número de procesadores es fijo y conocido**. Es decir, para el problema en que se desconocen las prioridades, el objetivo sería maximizar el número de tareas a ser ejecutadas en un número de máquinas fijo. En caso de conocer las prioridades, el objetivo es encontrar la máxima prioridad total del subconjunto de tareas seleccionado para un número de máquinas fijo. Ambos problemas se explican en el **Capítulo 3.1**.

  Dentro del caso con prioridades, se explican variantes como el ***Interval Scheduling* con tareas requeridas**, el ***Interval Scheduling* con disponibilidades de máquinas** o el ***Interval Scheduling* jerárquico**, en la **Sección 3.1.2**.

- Otro caso es el del ***Interval Scheduling Maximizing Minimum Coverage*** que se trata en el **Capítulo 3.2** y cuyo objetivo es seleccionar un subconjunto de tareas a ejecutar en un **número de máquinas dado**, de forma que el número mínimo de máquinas ocupadas en un momento dado sea el máximo posible.

- Otro de las posibles variantes consiste en, partiendo de que el **número de máquinas es variable y desconocido**, buscar el **número mínimo de máquinas** con las que se puedan ejecutar todas las tareas. Este problema se conocide como ***Interval Partitioning*** y se trata en el **Capítulo 3.3**.

Otro tipo de problemas son los conocidos como ***Online Interval Scheduling Problems***, [4], en los que las tareas tienen un tiempo de inicio y un tiempo de

finalización y se van presentando en orden creciente de su tiempo de inicio s_j, de forma que no existe una visión global del conjunto de tareas total. Se debe decidir por cada una si aceptarla o no, antes de recibir la siguiente.
El **objetivo** de estos problemas es maximizar la longitud total de tareas aceptadas, asegurando que ninguna de ellas se solape con otra.

La **variante con prioridades** de este problema permite eliminar tareas aceptadas previamente para seleccionar una nueva, eliminando con ella, su prioridad de la suma total.
En ese caso, el **objetivo** es maximizar la prioridad total de las tareas aceptadas y no eliminadas posteriormente.

Las soluciones para estos problemas no serán tratadas en profundidad: Woeginger muestra que se puede obtener una solución aceptable para la última variante con un algoritmo no determinista (ofrece muchos posibles resultados para la misma entrada).
En el caso sin prioridades con m máquinas, Faigle y Nawijn y de forma independiente Carlisle y Lloyd observaron que un algoritmo *greedy* es equivalente a uno *online* que siempre devuelve una solución óptima.
Erlebach y Spieksma estudiaron además la versión *online* del problema de *Interval Scheduling* discreto para el caso multiprocesador.

1.2. Variantes

A continuación, se describen de forma más detallada las diferentes variantes del problema de tareas con intervalos o *Interval Scheduling*.

1.2.1. Programación de tareas con intervalos con una máquina.

Puesto que solo se dispone de una máquina, es indispensable que las tareas seleccionadas no se solapen entre sí, de otra forma serían incompatibles. Esto es, que el tiempo de inicio de una tarea seleccionada no esté comprendido en el intervalo de procesamiento de otra de las tareas en el subconjunto elegido.
Formalmente, [**5**], dadas una tarea j y una tarea k, se exige que, si $f_j > s_k$ y $f_k > s_j$, entonces solo se seleccione una de ellas como máximo.

Interval Scheduling Maximization Problem: **Prioridades desconocidas.**

Este problema, conocido como ***Interval Scheduling Maximization Problem***, [**5**][**8**], es una simplificación del que se explica a continuación, *Weighted*

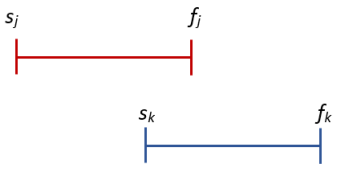

Figura 1.1: Solapamiento entre dos tareas j y k

Interval Scheduling Problem. Dicha simplificación se debe a que, en este caso, no se conoce el valor de la prioridad de cada tarea o todas tienen la misma prioridad. Es decir, se considera que el valor de p_j es 1 para toda tarea j; mientras que en el caso general, el valor de la prioridad es conocido y no tiene por qué ser equivalente para todas las tareas.

Bajo esa suposición, el objetivo es el siguiente: Maximizar $\sum_{j=1}^{n} x_j$.

De esta forma, lo óptimo será ejecutar el número máximo de tareas posible que no se solapen. Este problema se explica con más detalle en la **Sección 2.1**.

***Weighted Interval Scheduling Problem:* Conocidas las prioridades.**

En este caso, versión más genérica del anterior, se busca un subconjunto de tareas, y su correspondiente secuenciación, cuya suma de prioridades sea máxima. La formulación del problema, utilizando variables binarias, [**5**], sería:

Maximizar $\sum_{j=1}^{n} p_j x_j$

Sujeto a: $x_j + x_k \leq 1, \forall j,k \mid f_j > s_k \text{ y } f_k > s_j$ (1.1)

$x_j \in \{0,1\}\ j = 1, ..., n$ (1.2)

Donde p_j es la prioridad de la tarea j-ésima y x_j es 1 si se selecciona la tarea j-ésima y 0 en caso contrario.

La función objetivo establece que el valor a maximizar es la suma de prioridades. En cuanto a las restricciones, (1.1) indica que, en caso de que dos tareas se solapen, la suma de sus correspondientes variables binarias será como máximo 1, o lo que es lo mismo, solo se podrá seleccionar una de las dos tareas o ninguna, nunca las dos.
(1.2) establece que las variables que indican si una tarea es o no seleccionada podrán tomar valores del subconjunto {0,1}, es decir, son binarias.

Este problema es el conocido como ***Weighted Interval Scheduling Problem***, que parte de un conjunto de tareas con tiempos de inicio y de finalización para

cada tarea j-ésima, s_j y f_j respectivamente. En algunos casos, en lugar de hablar de prioridades, se conocen los beneficios de las tareas, b_j.
No obstante, esta diferencia es solo conceptual, ya que el objetivo y las restricciones no cambian de forma sustancial al hablar de beneficios o prioridades. Del mismo modo que se busca maximizar la suma de prioridades, se buscaría maximizar la suma de beneficios, por lo que para formular el problema sería suficiente con sustituir p_j por b_j.

Interval Scheduling with deadline and profits

En este caso se parte de los siguientes datos: el beneficio que genera la tarea j-ésima, b_j, y su *deadline*, d_j. Ha de tenerse en cuenta además, que el tiempo de procesamiento de cada tarea es unitario, lo que implica que cada tarea se ejecutará en el mismo tiempo que las *n-1* tareas restantes.
La ***deadline*** equivale al valor máximo que puede tomar el tiempo de finalización, f_j, de una tarea.
El tiempo de inicio de una tarea, s_j deberá ser, por tanto, menor que d_j en al menos una unidad de tiempo, de forma que f_j será igual o menor que d_j.

La formulación del problema según esto, sería:

Maximizar $\sum_{j=1}^{n} b_j x_j$

Sujeto a: $s_j + t_j \leq d_j + M(1 - x_j), \forall j = 1, ..., n$ (2.1)

$s_j + t_j - s_i \leq M y_{ij}, \qquad \forall i < j$ (2.2)

$s_i + t_i - s_j \leq M(1 - y_{ij}),$

$x_j \in \{0,1\}\ j = 1, ..., n$ (2.3)

$y_{i,j} \in \{0,1\}\ i, j = 1, ..., n$ (2.4)

$s_j \geq 1\ j = 1, ..., n$ (2.5)

La función objetivo busca maximizar el beneficio producido por el subconjunto de tareas ejecutado.
Por su parte, la restricción (2.1) indica que el valor de $s_j + t_j$, es decir, el valor de f_j, en caso de que una tarea sea ejecutada deberá ser menor que su *deadline*, d_j si el valor de x_j es 1, esto es, si la tarea se ejecuta; y menor que $d_j + M$, donde M es una cota superior suficientemente grande, en caso de que esta no se ejecute.
Las restricciones (2.2) buscan evitar los solapamientos entre dos tareas. Así, si dos tareas se solapan, es decir, si $y_{i,j} = 1$ para dos tareas i, j, de forma que $i \leq j$; entonces $s_j + t_j - s_i$ será menor o igual que M, mientras que $s_i + t_i$ será menor o

igual que s_j. Esto es, la tarea i-ésima, deberá terminar antes de que comience la tarea j-ésima.
En cuanto a las restricciones (2.3), (2.4) y (2.5), establecen que las variables que indican si una tarea j se ha ejecutado, x_j, y las variables que indican si dos tareas i, j se solapan, $y_{i,j}$, son binarias y que las variables de tiempo de inicio tomarán valores mayores o iguales a 1.

Interval Scheduling with deadline and release times

Los datos conocidos en este problema son: al igual que en el caso anterior, la *deadline*, d_j, de cada tarea y, además, su tiempo de salida o *release times*, r_j, o lo que es lo mismo, el momento a partir del cual podrá ser ejecutada la tarea j-ésima y el tiempo de procesamiento, p, que en este caso es el mismo para todas las tareas. El valor mínimo de r_j es 1 y el valor máximo de las *deadlines* se conoce como d_{max}. Es decir, todas las tareas ejecutadas lo hacen en el intervalo [1, d_{max}]. Se supone además que $d_j \geq r_j + p$ para todas las tareas.

Dado esto, el objetivo es maximizar el número de tareas a ejecutar.

Discrete Interval Scheduling Problem

Se trata, al igual que el problema de *Interval Scheduling with deadlines and release times*, de una variante [**4**], de los problemas de *scheduling* con restricciones de tiempo. En este caso, el valor de s_j puede tomar su valor de un conjunto S_j dado.

Minimize Maximum Lateness

En este caso, también se cuenta con el valor de la *deadline* de cada tarea, d_j, además de la duración de las mismas, t_j. Se definirá $f_j = s_j + t_j$ y el retraso de una tarea como $l_j = max(f_j - d_j, 0)$, de forma que esta se retrase si l_j es estrictamente positivo.
El objetivo es conseguir ejecutar todas las tareas de forma que el retraso máximo de la tarea dada, sea a su vez el mínimo posible.

La formulación del problema para esta variante sería:

Minimizar z

Sujeto a: $s_j + t_j - d_j \leq z$ (3.1)

$s_j + t_j - s_i \leq M y_{ij}, \quad \forall i < j$ (3.2)

$s_i + t_i - s_j \leq M(1 - y_{ij}),$

$$x_j \in \{0,1\}\ j = 1, ..., n \tag{3.3}$$

$$y_{i,j} \in \{0,1\}\ i, j = 1, ..., n \tag{3.4}$$

$$s_j \geq 1\ j = 1, ..., n \tag{3.5}$$

$$z \geq 0 \tag{3.6}$$

Es decir, se busca minimizar z, que será el valor máximo que pueda alcanzar $s_j + t_j - d_j$ para cualquier tarea j, es decir el valor máximo de las diferencias entre f_j y d_j, como se indica en la restricción (3.1).
El resto de restricciones, al igual que el caso del *Interval Scheduling with Deadlines and Profits* indican cuáles son las variables binarias, que el valor de s_j será mayor o igual a 1 para todas las tareas y que el valor del óptimo debe ser positivo.

Minimize Mean End Time

Este problema parte de un conjunto de tareas con tiempos de procesamiento t_j y prioridades p_j conocidos, siendo el objetivo minimizar la suma de los tiempos de procesamiento multiplicados por el valor inverso de la prioridad de cada tarea ejecutada.

1.2.2. Programación de tareas con intervalos con más de una máquina.

En este caso, las tareas seleccionadas pueden solaparse entre sí, siempre y cuando no se ejecuten en la misma máquina.
En cuanto a la clasificación, estos problemas pueden subdividirse en dos clases, aquellos en los que el **número de máquinas o procesadores es fijo y conocido** y aquellos en los que el **número de máquinas es variable o desconocido** y que por tanto forma parte del objetivo a conseguir.

Número de máquinas fijo y conocido.

Dentro de esta clasificación se encuentran las variantes correspondientes a la generalización de los **problemas básicos**: *Interval Scheduling Maximization Problem* y el *Weighted Interval Scheduling Problem* para varias máquinas o procesadores, así como las variantes para el caso con prioridades tratadas en la **Sección 3.1.2**: ***Interval Scheduling* con tareas requeridas**, ***Interval Scheduling* con disponibilidades de máquinas** e ***Interval Scheduling* jerárquico**. También en el caso del problema conocido como ***Interval Scheduling Maximizing Minimum Coverage*** el número de máquinas es fijo y conocido.

Casos básicos. Puesto que el número de máquinas, m, a utilizar es conocido, entonces el objetivo será, si la **prioridad es desconocida**, buscar el número máximo de tareas que es posible ejecutar en esos procesadores y **conocida la prioridad** de cada una de las tareas, encontrar la prioridad máxima total que se pueda obtener con dicho número de máquinas, como se describe brevemente en la referencia [**5**].

La formulación de estos modelos se diferencia de los casos básicos del *ISMP* y *WISP*, en que las variables binarias indicarán, además de la tarea de la que se trata, la máquina a la que está asignada. Esto se representa con un doble subíndice, de forma que x_{ij} tome el valor 1 si la tarea j-ésima está dentro del subconjunto de tareas asignadas a la máquina i-ésima y 0, en caso contrario.

Así, el modelo para el caso con prioridades, [**5**], sería:

$$\text{Maximizar } \sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{ij}$$

$$\text{Sujeto a: } x_{ij} + x_{ik} \leq 1,\ \forall i = 1, ..., m\ \forall j, k \mid f_j > s_k \text{ y } f_k > s_j \tag{4.1}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1,\ \forall j = 1, ..., n \tag{4.2}$$

$$x_{ij} \in \{0,1\}\ i = 1, ..., m,\ j = 1, ..., n \tag{4.3}$$

La función objetivo, al igual que en el caso anterior, busca maximizar la suma de prioridades y sería suficiente con que estas fueran unitarias, para representar el caso sin prioridades.
Las restricciones (4.1) y (4.3) se mantienen iguales que las restricciones (1.1) y (1.2) respectivamente, salvo por la adición del doble subíndice a las variables binarias.
Es necesario sin embargo, añadir una nueva restricción, (4.2), que exija que cada tarea, si es que se ejecuta, solo lo haga en una de las máquinas.

Al igual que en el caso en que solo se cuenta con una máquina o procesador disponible, cuando las prioridades son desconocidas, se puede obtener la solución óptima utilizando un algoritmo *greedy*, como se explicará más adelante.

Interval Scheduling Maximizing Minimum Coverage. Los datos de los que se parte en este problema son un conjunto de tareas con intervalos $[s_j,\ f_j)$ asociados, con el objetivo de seleccionar un subconjunto de tareas de forma que, al ejecutarse en un número de máquinas dado, el mínimo de las máquinas ocupadas en un momento concreto sea lo mayor posible.

Número de máquinas variable o desconocido, *Interval Partitioning.*

En el caso en que el número de máquinas no se conozca de antemano, el problema más conocido tendrá como objetivo optimizar el número de máquinas para realizar todas las tareas, es decir, buscar el número mínimo de procesadores que permite ejecutar el conjunto de tareas dado. Este problema es el conocido como ***Interval Partitioning***.

Para resolverlo, se parte de un número de máquinas suficientemente grande, m, a las que se hace referencia con las variables binarias, y_i, con $i = 1, ..., m$.

La formulación del modelo en este caso, sería:

Minimizar $\sum_{i=1}^{m} y_i$

Sujeto a: $\sum_{i=1}^{m} x_{ij} = 1 \ \forall j = 1, ..., n$ (5.1)

$x_{ij} \leq y_i \ \forall i = 1, ..., m, \ j = 1, ..., n$ (5.2)

$x_{ij} + x_{ik} \leq 1, \ \forall i = 1, ..., m \ \forall j, k \mid f_j > s_k \text{ y } f_k > s_j$ (5.3)

$x_{ij} \in \{0,1\} \ i = 1, ..., m, \ j = 1, ..., n,$ (5.4)

$y_i \in \{0,1\} \ i = 1, ..., m$ (5.5)

Como es de esperar, la función objetivo busca minimizar la suma de máquinas a utilizar.
Respecto a las restricciones, la restricción (5.1) indica que cada tarea debe ejecutarse y que por tanto, la suma de las x_{ij} será 1, de forma que se ejecutará en una y solo una de las m máquinas.
La restricción (5.2) implica que si la máquina i-ésima no va a ser utilizada, es decir, si y_i es 0, entonces todas las variables binarias x_{ij}, de forma que el primer subíndice de la x se corresponda con la máquina i-ésima, tomarán el valor 0. Es decir, solo podrán asignarse tareas a aquellas máquinas que estén en uso.
Por último, la restricción (5.3) es equivalente a la (4.1) y algo semejante ocurre con las restricciones (5.4) y (5.5), que indican que las variables a utilizar son binarias.

1.3. Aplicaciones

El uso de los problemas de *Interval Scheduling* viene motivado por su utilidad ante la aparición de un nuevo contexto, [**4**], ya que, en los últimos años, se ha producido una transición desde las logísticas **orientadas a recursos** a las logísticas

orientadas a la demanda. Es decir, la disponibilidad de recursos ha perdido fuerza frente a la necesidad de completar las tareas a la hora de obtener una planificación.
Este cambio ha venido generado ante la casi obligatoriedad, cada vez más imperante, de proporcionar un servicio al cliente cuando este lo solicite, al margen de la disponibilidad actual de recursos.

Concretamente, algunas de las áreas donde más uso tienen este tipo de problemas son:

- **Planificación de vehículos/tripulación.** El problema básico de planificación de tripulaciones puede formularse de la siguiente forma:
  Dado un conjunto de tripulaciones, otro de localizaciones y un conjunto de tareas; para **cada tarea** se da un tiempo de inicio, un tiempo de finalización y una localización y para **cada par de localizaciones** se tiene una distancia.
  El problema de asignar tareas a tripulaciones usando un número mínimo de las mismas puede formularse utilizando terminología típica del problema de *Interval Scheduling* de forma que, además de las tareas, **las tripulaciones sean las máquinas** y permitiendo una distancia entre cada par de tareas.
  Este tipo de problemas se dan también en aplicaciones para conductores de autobús y podrían asemejarse a la variante con **más de una máquina**, donde el **número** de las mismas sea **variable** o desconocido.

- **Telecomunicaciones.** En la situación en que varios usuarios se comunican entre sí usando una red, para que esto ocurra un usuario solicita la capacidad de un enlace o ancho de banda en la red durante un intervalo dado. Si varios usuarios solicitan esta capacidad a la vez, pueden darse incompatibilidades por falta de disponibilidad de ancho de banda. El objetivo en este caso será hacer un uso óptimo de la red asignando el ancho de banda disponible a los usuarios. En términos de *Interval Scheduling* las **solicitudes** son **tareas** y los **enlaces** o ancho de banda, **máquinas**.

- **Secuenciación de genoma y haplotipado**. En concreto cuando un conjunto de lecturas de ADN necesitan alinearse con un genoma para ser podados de forma que no más de k lecturas se solapen, manteniendo cubierta la máxima lectura posible a través del genoma completo. En este caso las lecturas serían tareas y el genoma las máquinas.

- **Otras aplicaciones.** Autores como Gabrel consideran el problema de **fotografías vía satélite**. El objetivo es tomar fotografías de la superficie de la Tierra alrededor de la que orbita. Fotografiar una zona específica de la Tierra implica que la cámara necesita empezar a filmar y dejar de hacerlo en determinados instantes. En términos de *Interval Scheduling*, fotografiar una zona concreta es una tarea y la cámara es una máquina.

Anthonisse y Lenstra describen otro problema relativo al **alquiler de cabañas**. Dado un conjunto de cabañas idénticas, el objetivo es responder a las reservas de los clientes lo más rápido posible. Dada la posibilidad de que algunos periodos de tiempo ya estén completos, el problema se convierte en una variante de *Interval Scheduling* donde las cabañas son las máquinas y las reservas son tareas.

Otro problema formulado por Gupta, Hashimoto y Stevens se refiere a la **asignación de canales**. Dado un par de componentes, cada una debe situarse en la coordenada y de un tablero y ambas deben estar situadas en la misma coordenada x. De esta forma, ambas componentes están conectadas mediante un canal. Los canales no pueden solaparse entre sí. Minimizar el número de canales puede resolverse mediante un problema de *Interval Scheduling* básico.

En la industria de la aviación, Kolen y Kroon plantearon un problema de **mantenimiento**. En concreto, los ingenieros debían llevar a cabo tareas de mantenimiento en los aviones. Existían diferentes tipos de aviones y cada ingeniero solo podía realizar su tarea en un tipo específico de avión si tenía licencia para dicho tipo. Las tareas tenían tiempos de inicio y finalización fijos. Formulado en términos de *Interval Scheduling*, los ingenieros serían las máquinas.

Otros autores como Carter y Tovey describen el problema de **asignación de clases**. El objetivo es asignar n clases a m habitaciones R_j con $j = 1, ..., m$ bajo diferentes restricciones y objetivos. Una de las variantes consideradas ordena las clases de forma que si una clase acepta la habitación R_j entonces aceptará también cualquier habitación R_k con $k > j$. En términos de *Interval Scheduling*, las clases son tareas y las habitaciones máquinas. Se trataría concretamente de un problema de *Interval Scheduling* jerárquico.

Las **políticas de reemplazo de cachés** fueron estudiadas por Brehob. Se pueden identificar con un problema de *Interval Scheduling* asociando cada acceso a un elemento concreto con un tiempo de inicio de una tarea y el siguiente acceso a ese elemento con su tiempo de finalización. Cada máquina se corresopnde con una localización en caché.

Otro problema es el de la **biología computacional**. Dada una secuencia de aminoácidos y de segmentos, y un beneficio para cada posible asignación de un segmento a una posición en la secuencia de aminoácidos, el objetivo es encontrar una asignación de forma que ningún segmento se solape y se maximice el beneficio total. En términos de *Interval Scheduling* la secuencia de aminoácidos será la máquina y los segmentos las tareas.

Capítulo 2

Tareas con un único procesador.

2.1. *Interval Scheduling Maximization Problem*

2.1.1. Descripción

Como se ha explicado en el apartado **Programación de tareas con intervalos con una máquina**, existen diferentes variantes del problema de *Interval Scheduling* con una máquina.

En concreto, para la variante de la sección ***Interval Scheduling Maximization Problem:* Prioridades desconocidas**, se desconocen las prioridades de cada tarea (o se consideran todas iguales a 1), se conocen los tiempos de inicio, s_j y f_j para un conjunto de n tareas y se tiene como **objetivo** maximizar el número de tareas a ejecutar, [**9**][**7**][**4**].
Este problema presentaba una **formulación** similar a la del *Weighted Interval Scheduling Problem* salvo por las prioridades unitarias, es decir:

Maximizar $\sum_{j=1}^{n} x_j$

Sujeto a: $x_j + x_k \leq 1, \forall j,k \mid f_j > s_k \text{ y } f_k > s_j$ (6.1)

$x_j \in \{0,1\}\ j = 1, ..., n$ (6.2)

Puede obtener su solución óptima bien de forma gráfica resolviendo el problema conocido como ***Maximum Independent Set*** o bien mediante un **algoritmo *greedy***, a diferencia de la variante en la que se conocen las prioridades de cada tarea.
Su solución óptima consiste en encontrar el número máximo de tareas que se pueden ejecutar en una máquina sin que sus tiempos de procesamiento se solapen.

De forma gráfica, cada tarea se consideraría un nodo en un gráfico. Si dos tareas son incompatibles, en este caso, si dos tareas se solapan, se representará de forma que ambas estén conectadas. La solución al problema es el mayor subconjunto de tareas compatibles, o lo que es lo mismo, el mayor conjunto de nodos independientes entre sí del gráfico.

A continuación se explica el **algoritmo *greedy*** en más detalle.

2.1.2. Algoritmo *greedy*

El **algoritmo *greedy*** se basa en considerar un orden de las tareas, que se van añadiendo a la selección siempre que sean compatibles con las ya elegidas.

Existen diferentes posibilidades, **[10][11][12][13]**, a la hora de considerar un orden para las tareas antes de iterar sobre ellas:

- **Menor tiempo de inicio:** Se ordenan por tiempo de inicio, s_j, ascendente.
- **Menor tiempo de finalización:** Se ordenan por tiempo de finalización, f_j, ascendente.
- **Menor tiempo de procesamiento:** Se ordenan por tiempo de procesamiento, $t_j = f_j - s_j$, ascendente.
- **Menor número de conflictos:** Para cada tarea j, se hace un recuento del número de tareas con las que es incompatible, c_j y se ordenan por c_j ascendente.

Podría pensarse que la solución óptima puede obtenerse seleccionando cualquiera de estas posibilidades. Sin embargo, a la hora de elegir entre una de ellas, se encuentran algunos contraejemplos que llevan a pensar que la mejor opción es ordenar las tareas por **tiempo de finalización** ascendente.

En el primer caso, **menor tiempo de inicio**, podría darse el caso de que la primera tarea elegida también termine la última, de forma que todas las tareas que se encuentran comprendidas entre sus tiempos de inicio y finalización, es decir, todas las demás, no podrían ejecutarse.

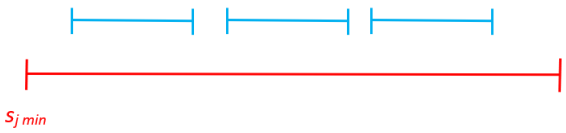

Figura 2.1: Contraejemplo menor s_j

En el tercer caso, **menor tiempo de procesamiento**, la tarea con menor tiempo de procesamiento podría solaparse con dos tareas más largas, de forma que solo se ejecutaría dicha tarea impidiendo que se ejecutasen las otras dos. Esto sería contraproducente para el objetivo, ya que se busca ejecutar el máximo número de tareas posible.

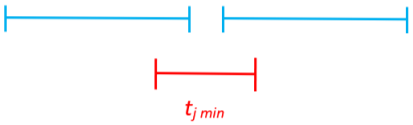

Figura 2.2: Contraejemplo menor t_j

En el cuarto caso, **menor número de conflictos**, podría encontrarse una tarea que entre en conflicto con menos tareas que el resto, pero que obligue a ejecutar aquellas que tienen más conflictos llevando de nuevo a la ejecución de un número menor de tareas. Sin embargo, puede darse el caso de que ejecutar una tarea con un mayor número de conflictos evite la ejecución de dichos conflictos, pero lleve a ejecutar otro subconjunto de tareas mayor.

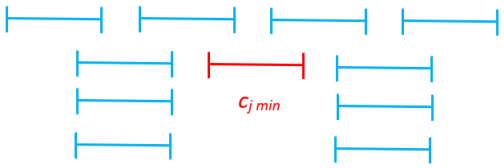

Figura 2.3: Contraejemplo menor c_j

Los **datos iniciales** son un conjunto de tareas, N, a ejecutar con tiempos de inicio y finalización, s_j y f_j respectivamente, para la tarea j-ésima y la **solución** que se busca es un subconjunto S, tal que S $\subseteq$ N.

El **pseudocódigo** básico, **[8][11][7][13]**, tomaría la siguiente estructura:

C = N, S = ϕ
while C $\neq \phi$ **do**
    S = S $\cup$ {k} donde k $\in$ C / $f_k = min\{f_j : j \in C\}$
    C = C - {{k} $\cup$ T} donde T contiene todas las tareas que se solapan con k.
end while
return S

Se puede mejorar si además se ordena el conjunto inicial N por tiempo de finalización, f_j, creciente, de forma que el **pseudocódigo**, [**9**][**10**], sería de la forma:

```
Ordenar N por f_j creciente
C = N, S = φ
for c in C do
    if c no se solapa con las tareas de S then
        S = S ∪ {c}
    end if
end for
return S
```

Es decir, se inicializa la lista de tareas candidatas con todas las tareas posibles (ordenadas o no) y la solución, que contendrá el número máximo de tareas que se puedan ejecutar en una sola máquina, como una lista vacía.

En la **primera opción**, en cada iteración, se comprueba que la lista de tareas candidatas contenga alguna tarea. Si es así, se busca aquella que menor tiempo de finalización, f_k, tenga y se selecciona dicha tarea. Se añade a la solución y se elimina de la lista de tareas candidatas, junto con el resto de tareas que se solapen con la seleccionada.

En la **segunda**, se recorre la lista de candidatas, ya ordenada por f_j creciente, y se seleccionan por orden todas aquellas tareas que no se solapen con las añadidas anteriormente.
Cuando la lista de candidatas se quede vacía o bien no exista ninguna tarea en la misma que no se solape con las contenidas en la solución (en caso de no eliminar a las candidatas que se vayan incluyendo en la solución), se habrá obtenido la solución óptima.

Prueba

Para probar que el algoritmo *greedy* proporciona siempre la solución óptima, [**9**][**10**][**11**][**7**], para este problema se parte de la segunda variante del pseudocódigo.
La prueba consiste en demostrar que para cada subproblema el algoritmo obtiene siempre la mejor solución posible, hasta llegar al subproblema mínimo.

Para ello, se supone la existencia de una solución S, obtenida con el algoritmo, y la solución óptima θ. El objetivo es probar que $|S| = |\theta|$.
Se supone también que ambos subconjuntos de tareas están ordenados por tiempo de finalización de las tareas en orden creciente.
En primer lugar, se probará que $f_j^S \leq f_j^\theta$, $\forall j$ mediante **inducción**. Esto demues-

tra que el algoritmo *greedy* siempre tendrá el mismo tiempo o más que la solución óptima para asignar a otras tareas posteriores a la tarea j-ésima.

Caso base Sea j = 0, tanto la solución para el subproblema del algoritmo *greedy* como la solución óptima contienen una única tarea.
Dado que el algoritmo *greedy* siempre elige como primera tarea la de menor tiempo de finalización, debe cumplirse que $f_0^S \leq f_0^\theta$. Puesto que esta es la tarea de N con menor tiempo f_j, no existe la posibilidad de que la solución óptima contenga otra tarea con un menor tiempo de finalización.

Paso de inducción Se supone que $f_{r-1}^S \leq f_{r-1}^\theta$ para probar $f_r^S \leq f_r^\theta$.
Para ello, se intenta demostrar lo contrario, es decir, $f_r^S > f_r^\theta$ y llegar a una **contradicción**. Se sabe que $s_r^\theta \geq f_{r-1}^\theta$, ya que la solución implica que las tareas no pueden ser incompatibles y por tanto, no pueden solaparse entre sí.
Dado que $f_{r-1}^S \leq f_{r-1}^\theta$, la tarea r-ésima del subconjunto de θ tampoco se solapará con ninguna de S (ya que $f_{r-1}^S \leq f_{r-1}^\theta \leq s_r^\theta$).

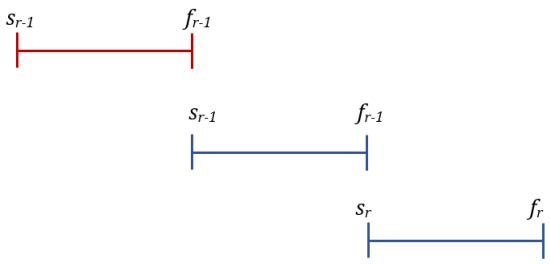

Figura 2.4: Representación gráfica de: $f_{r-1}^S \leq f_{r-1}^\theta \leq s_r^\theta$. (a)

(a) Las tareas de S se representan en rojo y las tareas de θ en azul.

Teniendo en cuenta esto y que $f_r^\theta < f_r^S$, el algoritmo habría elegido la tarea r-ésima de θ antes que la de S. Esto significaría que el algoritmo no ha elegido la tarea con el menor tiempo de finalización que no se solapase con el resto de tareas de S y por tanto, se llega a una contradicción y la hipótesis de que $f_r^S \leq f_r^\theta$ queda probada.

Aunque ahora se sabe que $f_r^S \leq f_r^\theta$, $\forall r$, esto no es suficiente para probar que el algoritmo proporciona siempre la solución óptima.
Este hecho, $(f_r^S \leq f_r^\theta, \forall r)$, se usará como lemma para terminar de probarlo.
Se sabe que $|S| = k$ y que $|\theta| = m$ y se desea probar que $k = m$. Dado que se sabe que θ es una solución óptima, no puede darse $k > m$, por lo que es suficiente con demostrar que $m \not> k$.
La demostración por **contradicción**, se basa en suponer que $m > k$.
Se sabe que para la tarea k - 1, $f_{k-1}^S \leq f_{k-1}^\theta$ y que θ tiene al menos una tarea más en su subconjunto, ya que $m > k$.
El algoritmo *greedy*, puesto que esta tarea k-ésima no se solapa con el resto, la

habría añadido al subconjunto S. Esto implica que $|S| \neq k$, lo que supone una contradicción.
Por tanto, la suposición $m > k$ es falsa y se puede afirmar que $k = m$. Dado que la solución *greedy* tiene el mismo número de tareas que la solución óptima, esta también debe ser óptima.

Complejidad

Partiendo de la segunda versión del pseudocódigo, se calcula la complejidad, [**9**], del mismo según lo siguiente:

1. La ordenación de las tareas por orden creciente de tiempo de finalización, puede realizarse con un algoritmo de ordenación, de los cuales, los más eficientes tienen una complejidad de $\boldsymbol{O(n \cdot log(n))}$.

2. Las asignaciones de valores a cualquier variable tienen una complejidad de $\boldsymbol{O(1)}$.

3. La complejidad del bucle es de $\boldsymbol{O(n)}$, siendo $|N| = n$. Las comprobaciones dentro del mismo tienen una complejidad de $\boldsymbol{O(1)}$.

   Por tanto, la complejidad total del algoritmo es de: $\boldsymbol{O(n \cdot log(n)) + O(n) + O(1)}$, lo que puede simplificarse a $\boldsymbol{O(n \cdot log(n))}$.

Implementación

Los **ficheros de datos** que se van a utilizar para probar la implementación del algoritmo *greedy* para este problema presentan la siguiente estructura: la primera línea contiene el número de tareas, n, que se va a leer. Las siguientes n líneas contienen un primer número entero: el índice asociado a la tarea, un segundo número, con el valor de s_j y un tercer número que contiene el valor de f_j para el índice correspondiente.
En concreto, en este caso se van a utilizar los ficheros: *interval_10.txt, interval_40.txt, interval_50.txt, interval_80.txt, interval_100.txt, interval_120.txt, interval_150.txt, interval_170.txt* e *interval_200.txt*.

En primer lugar, la implementación de **programación entera** utilizando *Xpress* presenta el aspecto de la **Figura 2.6**.

La primera línea se corresponde con la restricción que establece que las variables que indican si la tarea j-ésima se va a ejecutar o no, son binarias.
La segunda corresponde al valor del óptimo cuando se busca maximizar el número de tareas a ejecutar sin prioridades, es decir, se quiere maximizar la suma de tareas ejecutadas, o lo que es lo mismo: la suma de variables binarias con valor unitario.
La tercera línea establece la restricción según la cual si dos tareas se solapan, solo una de ellas podrá ejecutarse como máximo.

```
! MODELO DE PROGRAMACIÓN ENTERA: ISMP - Optimo

! x(j) en {0,1}
forall(j in tareas)x(j)is_binary

! Maximizar suma de x(j)
optimo:=sum(j in tareas)x(j)

! Ejecutar como maximo una tarea de aquellas que se solapen
forall(j,k in tareas|j<>k and f(j)>s(k) and f(k)>s(j))res(j,k):= x(j)+x(k)<=1

maximize(optimo)
```

Figura 2.6: ISMP - Problema de programación entera

(a) Código para obtener el óptimo con *Xpress*.

Por último, se indica que el objetivo es maximizar el valor óptimo.

En cuanto a la implementación del **algoritmo *greedy*** para obtener una solución al problema de maximización de *Interval Scheduling* que, como ya se ha indicado, devuelve el óptimo cuando no hay prioridades, tomaría la siguiente forma, en caso de ajustarse al segundo pseudocódigo de la sección 2.1.:

```
! Ordenar tareas por f(j) creciente
qsort(true, f, sorted_tareas)
writeln("\n\nTareas ordenadas segun f(j): \n")
writeln("\nj\ts\tf")
writeln("------------------")
forall(j in 1..n)writeln(sorted_tareas(j), "\t", s(sorted_tareas(j)), "\t", f(sorted_tareas(j)))
writeln("\n\n")
! CANDIDATAS = TAREAS
forall(j in tareas)candidatas(j):=1

! S = VACIO
forall(j in tareas)solucion(j):=0
opt:=0

! para c en CANDIDATAS: c = candidatas(j)
forall(j in tareas|candidatas(sorted_tareas(j))=1)do
    !Si c no se solapa con ninguna tarea de S
    if(opt>0)then
        forall(k in tareas|solucion(k)=1)do
            if(f(sorted_tareas(j))>s(k) and f(k)>s(sorted_tareas(j)))then
                candidatas(sorted_tareas(j)):=0
            end-if
        end-do
        if(candidatas(sorted_tareas(j))=1)then ! S = S + c
            writeln("Entra tarea: ", sorted_tareas(j), "\ts= ", s(sorted_tareas(j)), "\tf= ", f(sorted_tareas(j)))
            solucion(sorted_tareas(j)):=1
            opt:=opt+1
        end-if
    else
        opt:=opt + 1
        writeln("Entra tarea: ", sorted_tareas(j), "\ts= ", s(sorted_tareas(j)), "\tf= ", f(sorted_tareas(j)))
        solucion(sorted_tareas(j)):=1
        candidatas(sorted_tareas(j)):=0
    end-if
end-do
```

Figura 2.8: ISMP - Algoritmo *greedy*

(a) Código para obtener el óptimo con el algoritmo *greedy* en *Xpress*.

Como se indica en el mismo, el primer paso es ordenar las tareas por orden creciente de f_j, en este caso utilizando el procedimiento *qsort* del módulo *mmsystem*, [**14**], concretamente el que presenta la sintaxis:

```
procedure qsort(sense:boolean, vals: array of integer, lndx:list)
```

De forma que el valor de *sense* indique si el sentido de la ordenación será ascendente (*true*) o descendente (*false*), el argumento *vals* contenga los valores según los que se establecerá el orden, y *lndx* sea la lista donde se incluirán los índices del segundo argumento ordenados. Así, el índice correspondiente al valor más pequeño de f_j será el primer valor de la lista *lndx* y el correspondiente al valor más grande tomará el último puesto de *lndx*.
A continuación, se establece que todas las tareas son candidatas a ser ejecutadas y que la solución está actualmente vacía. Esto se consigue haciendo que el valor de los elementos de *candidatas* para cualquier índice j sea 1, es decir, tomando sus valores como binarios y lo contrario para los elementos de *solucion*.

Del mismo modo, se indica que el valor actual de *opt* es 0, ya que al principio no se ha añadido ninguna tarea a la solución. Este valor podría obtenerse también al final de la ejecución del algoritmo, sumando los elementos de *solucion* que presenten un valor unitario.

Lo siguiente es recorrer la lista de tareas que puedan ser ejecutadas, es decir, los elementos de *candidatas* que tomen el valor 1.
Puesto que en la primera iteración no se ha agregado ninguna tarea a la solución, es imposible que se encuentre alguna tarea que se solape con las de la solución. Por tanto, se agrega la primera de la lista de tareas ordenada según f_j a la solución y se elimina de la lista de candidatas.
En las siguientes iteraciones, se comprueba que las tareas que sigan siendo candidatas no se solapen con las ya contenidas en la solución, y en caso de ser compatibles, se agregan a la misma, dejando así de ser candidatas. En caso de que durante la ejecución se encuentre alguna tarea que se solape con alguna de las tareas de la solución, esta dejará de ser candidata a entrar en la misma.

Por último, una vez la lista de candidatas deje de tener elementos cuyo valor sea 1, se terminará la ejecución del algoritmo devolviendo el valor de *opt* y la lista de tareas a ejecutar que serán los elementos de *solucion*.

Soluciones obtenidas. A continuación se muestran los valores obtenidos para cada fichero de datos con la formulación de programación entera y el algoritmo *greedy*. Puesto que el algoritmo *greedy* encuentra el óptimo no se busca ninguna mejora en los resultados y simplemente se compara con el óptimo obtenido con *Xpress*:

Fichero de datos	Número de tareas	Óptimo	Solución *greedy*
interval_10.txt	10	5	5
interval_40.txt	40	6	6
interval_50.txt	50	17	17
interval_80.txt	80	23	23
interval_100.txt	100	15	15
interval_120.txt	120	7	7
interval_150.txt	150	8	8
interval_170.txt	170	8	8
interval_200.txt	200	7	7

Para mostrar un ejemplo de la salida obtenida con la ejecución del problema de programación entera y el algoritmo *greedy* se elige el fichero *interval_50.txt*, ya que, aunque el óptimo se corresponde con la solución obtenida con el algoritmo *greedy*, la solución del problema de programación entera devuelve un conjunto de tareas diferente, lo que muestra que en este caso existen al menos dos soluciones óptimas posibles.
Así, el conjunto óptimo de tareas a ejecutar según la solución del problema de programación entera, es el siguiente:

j	s_j	f_j
1	59	65
6	21	25
13	84	85
14	73	79
18	17	21
22	85	89
23	97	101
24	67	68
25	53	55
30	29	32
38	10	16
41	48	53
42	2	3
43	93	97
44	4	9
45	38	45
48	28	29

Mientras que la solución *greedy*, establece que el conjunto de tareas óptimo a ejecutar será:

j	s_j	f_j
6	21	25
13	84	85
14	73	79
15	62	64
18	17	21
22	85	89
23	97	101
24	67	68
25	53	55
30	29	32
37	41	50
39	11	12
42	2	3
43	93	97
44	4	9
46	32	39
48	28	29

Esto se debe a que, por ejemplo, la tarea 15 presenta un valor de f_j menor que el de la tarea 1 y por tanto, es añadida a la solución *greedy* antes, dado que la tarea 15 precedería a la 1 en la lista de tareas ordenadas por f_j. Sin embargo, el número de tareas máximo sigue siendo el mismo.
De hecho, puede observarse cómo el orden de entrada a la solución de las tareas, en la ejecución del algoritmo *greedy* es por orden de f_j, lo que llevaría a estas diferencias en los elementos que forman ambas soluciones:

```
Entra tarea: 42 s= 2    f= 3
Entra tarea: 44 s= 4    f= 9
Entra tarea: 39 s= 11   f= 12
Entra tarea: 18 s= 17   f= 21
Entra tarea: 6  s= 21   f= 25
Entra tarea: 48 s= 28   f= 29
Entra tarea: 30 s= 29   f= 32
Entra tarea: 46 s= 32   f= 39
Entra tarea: 37 s= 41   f= 50
Entra tarea: 25 s= 53   f= 55
Entra tarea: 15 s= 62   f= 64
Entra tarea: 24 s= 67   f= 68
Entra tarea: 14 s= 73   f= 79
Entra tarea: 13 s= 84   f= 85
Entra tarea: 22 s= 85   f= 89
Entra tarea: 43 s= 93   f= 97
Entra tarea: 23 s= 97   f= 101
```

Figura 2.10: ISMP - Algoritmo *greedy*

(a) Ejecución del alogritmo *greedy* para el fichero *interval_50.txt*.

2.2. *Weighted Interval Scheduling Problem*

2.2.1. Descripción

La variante genérica del problema presentado en la sección anterior es aquella en la que las tareas, además de un tiempo de inicio y otro de finalización, tienen prioridades no unitarias.

Así, los **datos iniciales** de este problema, [**3**], son un conjunto de N tareas con tiempo de inicio s_j, tiempo de finalización f_j y prioridad p_j. Se define además com_j como el máximo valor del índice $i < j$ que sea compatible con la tarea j.
El **objetivo** es encontrar un subconjunto de tareas con prioridad total máxima y compatibles entre sí.

La **formulación** en este caso, como ya se ha explicado en la sección ***Weighted Interval Scheduling Problem:* Conocidas las prioridades** es la siguiente:

$$\text{Maximizar } \sum_{j=1}^{n} p_j x_j$$

$$\text{Sujeto a: } x_j + x_k \leq 1,\ \forall j,k \mid f_j > s_k \text{ y } f_k > s_j \qquad (7.1)$$

$$x_j \in \{0,1\}\ j = 1, ..., n \qquad (7.2)$$

2.2.2. Métodos para buscar soluciones

En esta sección se van a explicar diferentes formas de obtener soluciones para el problema de *Interval Scheduling* con prioridades en un único procesador, concretamente el algoritmo *greedy* con la aplicación posterior de una heurística de mejora y un algoritmo de fuerza bruta basado en la elección binaria.
A pesar de hablar de otros métodos además de este, solo se implementará el algoritmo *greedy* junto con la heurística.

Algoritmo *greedy*

En el caso en que no hay prioridades o todas tienen valor unitario, el algoritmo *greedy* devuelve la solución óptima del problema.
En esa situación, el algoritmo *greedy* consiste en ordenar las tareas en orden creciente de tiempo de finalización f_j e ir ejecutando las tareas según dicho orden siempre que sean compatibles con las que ya han sido ejecutadas.
Sin embargo, si se permiten prioridades arbitrarias dicho algoritmo puede obtener resultados muy malos si, por ejemplo, siguiendo los principios en los que se basan

los algoritmos *greedy*, se decide ordenar las tareas en función de su prioridad, de forma decreciente, optando primero por aquellas tareas con un valor mayor de su prioridad, p_j.
En ese caso, puede ocurrir que una sola tarea tenga un valor de la prioridad ligeramente más alto e impida que se ejecuten el resto aunque estas representen un subconjunto de tareas compatible cuya suma de prioridades sea mayor.

A pesar de esto, es posible utilizar la solución *greedy* para acercarnos al óptimo, jugando en este caso su papel de heurística de construcción.
Los **métodos heurísticos**, a los que se está haciendo referencia al hablar de heurística de construcción, se desarrollan como consecuencia de las dificultades a la hora de resolver problemas de forma óptima, bien por su magnitud o por falta de recursos u otras circunstancias, proporcionando una solución aproximada y factible que a pesar de no optimizar el resultado, permita acercarse al valor óptimo en un tiempo razonable.
Dentro de estos métodos heurísticos o heurísticas, se puede diferenciar entre **heurísticas de construcción** y **heurísticas de mejora**. El aquí mencionado algoritmo *greedy* pertenece a la primera clasificación y es de los métodos constructivos más utilizados. Estos permiten obtener, paso a paso, una solución factible del problema, que puede ser o no la solución óptima.
Por su parte, las heurísticas de mejora parten de una solución factible, como podría ser, por ejemplo, la obtenida con el algoritmo *greedy* y buscan una solución mejor.

Es decir, en aquellos casos en los que el algoritmo *greedy* no permite obtener una solución óptima, dado que su implementación no admite reorganizaciones en la solución una vez tomada una decisión, sí que facilita una solución factible a partir de la cual se pueden obtener mejores resultados aplicando una heurística de mejora.

Por tanto, para el *Weighted Interval Scheduling Problem*, se puede utilizar la solución *greedy* con alguna heurística de mejora para conseguir una solución más cercana o igual a la óptima.
En este caso, se optará por aplicar la **búsqueda local** como heurística de mejora. Es una de las heurísticas de mejora más usadas y se basa en realizar movimientos sobre el entorno de la solución actual, obteniendo nuevas soluciones, que puedan suponer o no una mejora. El tipo de movimientos elegidos que definirán el entorno de la solución en este caso se explica en la implementación de esta heurística.
Si solo se aceptan soluciones que supongan una mejora, se habla de **método de descenso**. Concretamente, se hablará en este apartado del mayor descenso o ***best improvement***, que examina el entorno de la solución actual por completo para conseguir la mejor solución dentro de dicho entorno. En casos en los que el entorno de la solución es demasiado grande, se puede optar por elegir la primera solución que suponga una mejora o *first improvement* o bien por elegir soluciones del entorno aleatoriamente o descenso estocástico.
En casos más complejos, este método podría no ser suficiente, dado que al buscar siempre una solución mejor suele estancarse en un óptimo local, a pesar de mejorar

los resultados obtenidos con la heurística de construcción, impidiendo así alcanzar el óptimo global.
Para resolver este problema, se utilizan métodos heurísticos más complejos o metaheurísticas, que no solo permiten movimientos que lleven a una solución mejor, como es el caso de las heurísticas de mejora, y en consecuencia su alcance es más amplio.

Implementación A continuación se muestran las soluciones obtenidas con el algoritmo *greedy* para los siguientes **ficheros de datos**: *prioridades_10.txt, prioridades_50.txt, prioridades_70.txt, prioridades_100.txt, prioridades_120.txt, prioridades_140.txt, prioridades_160.txt, prioridades_180.txt, prioridades_200.txt* y *prioridades_300.txt* así como la mejora conseguida tras aplicar la heurística de mejora y la solución óptima obtenida con *Xpress*.
La estructura de los ficheros de datos es similar a la de los utilizados para el problema anterior: en la primera línea se encuentra el número de tareas a ejecutar, en las siguientes n líneas se sigue el mismo formato, un primer número con el índice de la tarea, el siguiente número que identifica el valor de s_j, el tercero correspondiente a f_j y un cuarto número que se corresponde con la prioridad, p_j de la tarea.

La implementación de **programación entera** sería igual que en el caso del problema de ISMP, salvo por la función objetivo, es decir:

```
! MODELO  DE PROGRAMACIÓN ENTERA: WISP - Óptimo

! x(j) en {0,1}
forall(j in tareas)x(j)is_binary

! Maximizar suma de p(j)*x(j) - Prioridad total
optimo:=sum(j in tareas)p(j)*x(j)

! Ejecutar como maximo una tarea de aquellas que se solapen
forall(j,k in tareas|j<>k and f(j)>s(k) and f(k)>s(j))res(j,k):= x(j)+x(k)<=1

maximize(optimo)
```

Figura 2.12: WISP - Problema de programación entera

(a) Código para obtener el óptimo con *Xpress*.

Al igual que en el caso anterior, las variables de decisión son binarias y en caso de que se solapen solo podrá ejecutarse una tarea como máximo. Por su parte, el valor óptimo a maximizar se calcula en este caso como la suma de las prioridades de aquellas tareas que se vayan a ejecutar. Por tanto, para maximizar esta suma, deberán ejecutarse aquellas tareas compatibles entre sí que en conjunto sumen una prioridad mayor.

En cuanto al procedimiento correspondiente al **algoritmo *greedy***, se utiliza el

mismo que en el caso sin prioridades para obtener una solución factible, salvo porque en este caso para calcular el óptimo, se suma la prioridad correspondiente:

```
! Ordenar tareas por f(j) creciente
qsort(true, f, sorted_tareas)
writeln("\n\nTareas ordenadas segun f(j): \n")
writeln("\nj\ts\tf\tp")
writeln("-------------------------")
forall(j in 1..n)writeln(sorted_tareas(j), "\t", s(sorted_tareas(j)), "\t", f(sorted_tareas(j)), "\t", p(sorted_tareas(j)))
writeln("\n\n")

! CANDIDATAS = TAREAS
forall(j in tareas)candidatas(j):=1

! S = VACIO
forall(j in tareas)solucion(j):=0
opt:=0

! para c en CANDIDATAS: c = candidatas(j)
forall(j in tareas|candidatas(sorted_tareas(j))=1)do
    !Si c no se solapa con ninguna tarea de S
    if(opt>0)then
        forall(k in tareas|solucion(k)=1)do
            if(f(sorted_tareas(j))>s(k) and f(k)>s(sorted_tareas(j)))then
                candidatas(sorted_tareas(j)):=0
            end-if
        end-do
        if(candidatas(sorted_tareas(j))=1)then !S = S + c
            writeln("Entra tarea: ", sorted_tareas(j), "\ts= ", s(sorted_tareas(j)), "\tf= ", f(sorted_tareas(j)), "\tp= ", p(sorted_tareas(j)))
            solucion(sorted_tareas(j)):=1
            opt:=opt + p(sorted_tareas(j))
        end-if
    else
        opt:= opt + p(sorted_tareas(j))
        writeln("Entra tarea: ", sorted_tareas(j), "\ts= ", s(sorted_tareas(j)), "\tf= ", f(sorted_tareas(j)), "\tp= ", p(sorted_tareas(j)))
        solucion(sorted_tareas(j)):=1
        candidatas(sorted_tareas(j)):=0
    end-if
end-do
```

Figura 2.14: WISP - Algoritmo *greedy*

(a) Código para obtener una solución factible con el algoritmo *greedy* en *Xpress*

Al igual que en el problema sin prioridades, se ordenan las tareas por f_j creciente; se establecen todas las tareas como candidatas y se inicializa la solución con cero tareas a ejecutar y un óptimo nulo.
A la hora de recorrer las tareas candidatas, se agrega la primera de las tareas por orden creciente de f_j a la solución y se suma su prioridad al valor de *opt*, que inicialmente es cero.
Después, se recorre el resto de tareas hasta que dejen de encontrarse candidatas, eliminando aquellas que se solapen con las ya contenidas en la solución y añadiendo a la misma aquellas que en el momento en que se comprueben no se solapen con las elegidas hasta entonces. Al mismo tiempo que se añada una tarea, se suma al valor óptimo su prioridad, obteniendo al final la suma de prioridades de las tareas contenidas en la solución.

Como se ha indicado antes, el algoritmo *greedy* no puede obtener siempre el óptimo en este caso, dado que no se tiene en cuenta la prioridad de la tarea en el momento de añadirla a la solución, sino solo el valor de f_j y su compatibilidad con las tareas ya elegidas. Si en lugar de ordenarse las tareas por f_j creciente, se ordenasen por p_j decreciente también podría obtenerse una solución factible, que, como se ha indicado al inicio de la sección, tampoco tendría por qué ser la óptima, pues podría darse el caso de que una tarea con prioridad ligeramente más alta, evitase la ejecución de otra con prioridad parecida que permitiese la ejecución de otras tareas, de forma que la suma total de prioridades fuese mayor.

Es posible aproximarse al óptimo utilizando la búsqueda local como heurística de mejora. Además de optar por *best-improvement*, se ha escogido como definición de entorno o movimiento realizar dos tipos de intercambios. En un primer caso, se realiza un intercambio entre una tarea contenida en la solución y otra que no esté en la misma y en un segundo caso, se opta por sacar dos tareas de la solución y elegir otra que no estuviese en la solución actual.

La implementación de ambas opciones es similar. La **primera opción** de dos intercambios y *best-improvement* tendría el aspecto que se muestra en la Figura 2.16 en *Xpress*.

```
! BUSQUEDA LOCAL: 2 Intercambios + Best Improvement

procedure busqueda_local_2int
    final:=0
    while(final=0)do
        mejora_max:=-999
        forall(j, k in 1..n|solucion(j)=1 and solucion(k)=0)do
            xl(j):=0
            xl(k):=1
            forall(i in 1..n|i<>j and i<>k)xl(i):=solucion(i)
            if(factible(xl))then
                nuevo_objetivo:=calcular_objetivo(xl)
                writeln("\nLa nueva prioridad total es ", nuevo_objetivo)
                antiguo_objetivo:=sum(z in tareas)p(z)*solucion(z)
                writeln("La vieja prioridad total es ", antiguo_objetivo)
                mejora:=nuevo_objetivo - antiguo_objetivo
                writeln("La mejora es de ", mejora)
                if(mejora > mejora_max)then
                    mejora_max:= mejora
                    jmax:=j
                    kmax:=k
                end-if
            end-if
        end-do
        if(mejora_max <=0)then
            final:=1
        else
            solucion(jmax):=0
            solucion(kmax):=1
        end-if
    end-do

end-procedure
```

Figura 2.16: WISP - Búsqueda local con 2 intercambios y *best-improvement.*

(a) Código para obtener una solución mejor a partir de la solución *greedy* y búsqueda local.

La **segunda opción** como se ha indicado, intercambia dos tareas de la solución por otra que no esté en la misma, como puede verse en la Figura 2.18.

En ambas implementaciones se inicializa una variable bandera o $flag$, $final$, a cero, para indicar que la ejecución del procedimiento no ha terminado.
De el valor de esta variable dependerá la condición del bucle $while$ que se ejecutará a continuación.
Dentro del mismo, se inicializa la variable $mejora\_max$ a un valor suficientemente pequeño para que no suponga un problema a la hora de comprobar si la solución nueva es mejor que la anterior.

```
! BUSQUEDA LOCAL: 3 Intercambios + Best Improvement

procedure busqueda_local_3int
    final:=0
    while(final=0)do
        mejora_max:=-999
        forall(j, k, l in 1..n|solucion(j)=1 and solucion(k)=1 and solucion(l)=0)do
            x1(j):=0
            x1(k):=0
            x1(l):=1
            forall(i in 1..n|i<>j and i<>k and i<>l)x1(i):=solucion(i)
            if(factible(x1))then
                nuevo_objetivo:=calcular_objetivo(x1)
                writeln("\nLa nueva prioridad total es ", nuevo_objetivo)
                antiguo_objetivo:=sum(z in tareas)p(z)*solucion(z)
                writeln("La vieja prioridad total es ", antiguo_objetivo)
                mejora:=nuevo_objetivo - antiguo_objetivo
                writeln("La mejora es de ", mejora)
                if(mejora > mejora_max)then
                    mejora_max:= mejora
                    jmax:=j
                    kmax:=k
                    lmax:=l
                end-if
            end-if
        end-do
        if(mejora_max <=0)then
            final:=1
        else
            solucion(jmax):=0
            solucion(kmax):=0
            solucion(lmax):=1
        end-if
    end-do

end-procedure
```

Figura 2.18: WISP - Búsqueda local con 3 intercambios y *best-improvement.*

(a) Código para obtener una solución mejor a partir de la solución *greedy* y búsqueda local.

Es en el primer bucle donde se establece la diferencia entre una opción y la otra. En el **primer caso**, se seleccionan dos tareas, de forma que una esté en la solución y la otra no. Es decir, una tarea cuyo índice en el *array solucion* tome el valor 1 y otra que tome el valor 0.
En otro *array* con el mismo formato que el de *solucion*, en este caso $x1$, se intercambian los valores de ambos índices, de forma que si la tarea j-ésima toma el valor 1 en el *array solucion*, tome el valor 0 en el array $x1$ y si la tarea k-ésima toma el valor 0 en el *array solucion*, tome el valor 1 en $x1$. El resto de valores de $x1$ tomarán el mismo valor que tienen en *solucion*.

En el **segundo caso**, se seleccionan tres tareas, dos pertenecientes a la solución actual y una que no esté en la misma.
En este caso, el array $x1$ pasará a incluir en la solución a la tercera tarea y extraerá las dos primeras, que sí que estarían en la solución actual. Al igual que en el primer caso, el resto de valores de $x1$ serán equivalentes a los de *solucion*.

Después de esto, en ambos casos se comprueba si la solución obtenida en $x1$ es factible. En caso de que lo sea, se calcula el valor de la función objetivo, es decir, la

suma de prioridades de las variables seleccionadas con la nueva solución contenida en $x1$ y con la antigua solución, contenida en la variable *solucion*. Una vez se tienen ambos valores, se obtiene la diferencia entre el nuevo valor de la función objetivo y el actual o antiguo. Si el valor de esta diferencia, contenida en la variable $mejora$, es superior al de la variable $mejora\_max$, se habrá encontrado una solución mejor a la seleccionada con anterioridad, modificando el valor de la variable $mejora\_max$ y guardando los índices de las tareas intercambiadas que han proporcionado dicha mejora; es decir, en el primer caso, los de dos tareas y en el segundo, los de tres.

Finalmente, si la mejora obtenida no es positiva y ya se han recorrido todas las posibles opciones, se cambia el valor de la variable $flag$ dando por finalizada la ejecución del procedimiento. En otro caso, se modifican los valores de los índices seleccionados que han proporcionado el valor actual de $mejora\_max$ en la variable *solucion* y se continúa con la búsqueda local, realizando los intercambios sobre la nueva solución seleccionada.

Soluciones obtenidas. A continuación se muestran los valores obtenidos para cada fichero de datos con la formulación de programación entera, que nos asegura el óptimo, el algoritmo *greedy*, la búsqueda local con 2 intercambios y la búsqueda local con 3 intercambios:

Fichero de datos	Nº tareas	Óptimo	Sol. *greedy*	BL 2 ints	BL 3 ints
prioridades_10.txt	10	27	21	27	27
prioridades_50.txt	50	44	30	42	44
prioridades_70.txt	70	45	39	39	39
prioridades_100.txt	100	57	50	57	57
prioridades_120.txt	120	62	42	62	62
prioridades_140.txt	140	55	37	48	49
prioridades_160.txt	160	53	41	49	49
prioridades_180.txt	180	66	51	66	66
prioridades_200.txt	200	58	37	50	54
prioridades_300.txt	300	62	40	54	57

Se observa que tanto para el caso en que el número de tareas a ejecutar es 10 como en los casos en que son 100, 120 y 180, se alcanza el valor óptimo tras aplicar búsqueda local con 2 y 3 intercambios a la solución *greedy*. Sin embargo, en los otros casos, con búsqueda local con 2 intercambios no se consigue alcanzar el óptimo en ninguno y solo en el caso con 50 tareas se consigue aplicando búsqueda local con 3 intercambios.
La solución al problema para el fichero *prioridades_70.txt* parece quedarse en un óptimo local obtenido ya con la solución *greedy*.

Programación dinámica: elección binaria, [1][2][3]

La solución óptima hasta la tarea j-ésima puede tomar dos formas:

- **Se selecciona la tarea** j En ese caso no pueden incluirse en la solución las tareas $com_j + 1$, $com_j + 2$,..., j - 1, solo aquellas tareas compatibles con j hasta com_j incluida.
- **No se selecciona la tarea** j Se deben incluir en la solución aquellas tareas hasta la tarea $j - 1$ que sean compatibles.

Expresado de forma recursiva el valor de la solución óptima hasta la tarea j-ésima, $O(j)$ sería:

$$\text{O(j)} = \begin{cases} 0 & \text{si } j \text{ es } 0 \\ max\{p_j + O(com_j), O(j-1)\} & \text{en otro caso} \end{cases}$$

Algoritmo de fuerza bruta El algoritmo de fuerza bruta que implementa la forma recursiva de obtener la solución, presenta el siguiente **pseudocódigo**:

Ordenar las tareas por orden creciente de f_j
Calcular el valor de com_j $\forall j$
calcularOptimo(j)

Donde la implementación de **calcularOptimo(j)** devolvería:

if $j = 0$ **then return** 0
else return max{p_j + calcularOptimo(com_j), calcularOptimo(j-1)}
end if

Sin embargo, los algoritmos recursivos pueden ver afectado su resultado por problemas redundantes que generen un número de llamadas recursivas de manera exponencial.
Para solucionare esto se utiliza la recursión relajada, sustituyendo la llamada a **calcularOptimo(j)** por una llamada a la siguiente función, **calculoIterativo**, con el siguiente **pseudocódigo**:

calculoIterativo {
$opt_0 = 0$
for $j \in N$ **do**
  $opt_j = \max(p_j + opt_{com_j}, opt_{j-1})$
end for
} **return** opt_n

La **complejidad** de esta solución es de ***O(n · log(n))***, ya que el bucle principal tiene una complejidad de ***O(n)*** y ordenar las tareas por tiempo de finalización tiene como mínimo una complejidad de ***O(n · log(n))***.
En este caso, ordenar las tareas no tiene el mismo objetivo que en la implementación de los algoritmos *greedy*, sino que busca reducir el número de subproblemas generados en las llamadas recursivas.

2.3. *Interval Scheduling with Deadline and Profits*

2.3.1. Descripción

Los **datos iniciales** de este problema son un conjunto de tareas, N, con *deadline* d_j, beneficios b_j y duración t_j unitaria, [**15**][**7**][**16**].
La ***deadline***, d_j es, en este caso, el valor máximo que puede tomar f_j.

En los problemas en los que se conoce el valor de la *deadline* o fecha/momento límite, pueden darse dos situaciones: bien que, conocido este valor, solo se permita ejecutar la tarea, si su tiempo de finalización no sobrepasa el de d_j, o bien que, se permita sobrepasar dicho valor, pero que sea perjudicial para la solución del problema, es decir, que suponga cierta penalización sobre el valor a optimizar.

El **objetivo** en el caso del problema que se trata en esta sección, es ejecutar un subconjunto de tareas cuyo beneficio total sea máximo antes de sus correspondientes *deadlines* en una misma máquina, de forma que ninguna tarea del subconjunto sea incompatible con el resto.

La **formulación** del problema como se ha indicado antes, sería:

Maximizar $\sum_{j=1}^{n} b_j x_j$

Sujeto a: $s_j + t_j \leq d_j + M(1 - x_j), \forall j = 1, ..., n$ (8.1)

$s_j + t_j - s_i \leq M y_{ij}, \quad \forall i < j$ (8.2)

$s_i + t_i - s_j \leq M(1 - y_{ij}),$

$x_j \in \{0,1\}\ j = 1, ..., n$ (8.3)

$y_{i,j} \in \{0,1\}\ i, j = 1, ..., n$ (8.4)

$s_j \geq 1\ j = 1, ..., n$ (8.5)

Otro caso, en el cual los datos que se conocen del conjunto de tareas son: su *deadline*, d_j y su tiempo de procesamiento, t_j, pero no los beneficios de las mismas; presenta otro objetivo posible dado el valor de las *deadlines*: minimizar el retraso máximo o ***maximum lateness*** como se explica en la **Sección 2.6**.

2.3.2. Algoritmo *greedy*

El algoritmo para resolver el problema que busca optimizar los beneficios sin sobrepasar el valor de las *deadlines* se basa en el **procesamiento diferido** o ***Deffered Processing***, es decir, en ejecutar las tareas lo más tarde que se pueda, de forma que se permita ejecutar antes tareas con una *deadline* más temprana. Aquellas tareas que no puedan ejecutarse de esta forma, no serán seleccionadas.
Así, el **algoritmo *greedy*** utilizado para resolver este problema, [**15**], presenta el siguiente **pseudocódigo**:

Ordenar las tareas en orden decreciente de b_j
Crear un array S donde se irán almacenando los intervalos de tiempo de la máquina que continúen disponibles
for $j \in N$ **do**
  if Existe un intervalo de tiempo libre para la tarea j **then**
    Asignar el intervalo cuyo f_j sea más cercano a d_j a la tarea j, a ser posible una unidad de tiempo antes de su *deadline*
  else
    No asignar j
  end if
end for
return $[s_j,\ f_j]\ \forall j$

Complejidad

La **complejidad** de este algoritmo, considerando que se busca una secuenciación de las tareas, sería la siguiente:

- Si se usa un bucle en el **paso 3**, este tendría una complejidad de $O(n^2)$
- Puesto que en el **paso 1** es necesario utilizar un algoritmo de ordenación, la complejidad mínima necesaria para esto es de $O(n \cdot log(n))$, por ejemplo utilizando el algoritmo *quicksort*.

En conclusión, la complejidad del algoritmo sería de $O(n^2 + n \cdot log(n))$.

Implementación

A continuación se muestran las soluciones obtenidas con *Xpress* y con el algoritmo *greedy* para los siguientes **ficheros de datos**: *deadprofit_10.txt*, *deadprofit_20.txt*, *deadprofit_40.txt*, *deadprofit_60.txt*, *deadprofit_80.txt*, *deadprofit_100.txt*, *deadprofit_120.txt*, *deadprofit_140.txt*, *deadprofit_160.txt*, *deadprofit_180.txt* y *deadprofit_200.txt*.
Estos ficheros presentan la siguiente estructura: en la primera línea se lee el número de tareas a ejecutar, n, en las siguientes n líneas, se encuentra un primer número, el índice correspondiente a la tarea de esa línea, seguido del valor de la *deadline* y del beneficio de la tarea.

Se pueden encontrar problemas similares a estos en las referencias: [17][18].

La implementación de **programación entera** con *Xpress* es la siguiente:

```
! MODELO  DE PROGRAMACIÓN ENTERA: ISDP - Óptimo

M:=9999

! x(j) en {0,1}
forall(j in tareas)x(j)is_binary
! y(i,j) en {0,1}
forall(i,j in tareas)y(i,j)is_binary

! Maximizar suma de b(j)*x(j) - Beneficio total
optimo:=sum(j in tareas)b(j)*x(j)

! El tiempo de inicio debe ser menor al menos en una ud.
! que la deadline: tiempo de procesamiento unitario

forall(j in tareas)res_t(j):=init(j) + 1 <=d(j) + M*(1-x(j))

! Restricción de solapamiento

forall(i,j in tareas | i<j)res_21(i,j):= init(j) + 1 - init(i) <= M*y(i,j)
forall(i,j in tareas | i<j)res_22(i,j):= init(i) + 1 - init(j) <= M*(1-y(i,j))

! s(j)>= 1
forall(j in tareas)res(j):=init(j)>=1

! setparam("XPRS_MAXTIME", -240)

maximize(optimo)
```

Figura 2.20: ISDP - Problema de programación entera

(a) Código para obtener una solución con *Xpress*

En primer lugar, se asigna a M un valor suficientemente grande y se establecen las variables x_j e $y_{i,j}$ como binarias.
Después, se establece el valor del óptimo como la suma de los beneficios de las tareas que se vayan a ejecutar.
La primera restricción establece que la suma de s_j (renombrado como $init(j)$) con el tiempo de procesamiento, que en este caso es unitario será menor o igual que la

suma de $d_j + M(1 - x_j)$, de forma que en caso de que la tarea se ejecute, el valor de f_j, es decir, $s_j + 1$, no sea mayor que el de d_j.
Las siguientes restricciones evitan que las tareas ejecutadas se solapen, de forma que si dos tareas se solapan, la primera tome un valor de f_i, es decir, la suma de $init_i + 1$, que no sea mayor que el de $init_j$.
Por último, se establece que el valor de s_j, en el código $init_j$, es mayor o igual que 1 para todas las tareas.
Finalmente, se indica que se quiere maximizar el óptimo.

La implementación del **algoritmo *greedy*** en *Xpress* presenta el aspecto que se muestra en la **Figura 2.22**.

```
!Ordenar tareas por b(j) decreciente
qsort(false, b, sorted_tareas)

!f(j) DISPONIBLE
max_d:=max(j in tareas)d(j)
forall(i in 2..max_d)disponible(i):=1

forall(j in tareas)do
    f(sorted_tareas(j)):= buscar_intervalo(j)
    if(f(sorted_tareas(j))=-1)then
        solucion(sorted_tareas(j)):=0
    else
        solucion(sorted_tareas(j)):=1
        opt:=opt + b(sorted_tareas(j))
        ! El valor de f(sorted_tareas(j)) no está disponible
        indice:=f(sorted_tareas(j))
        disponible(indice):=0
        s(sorted_tareas(j)):=f(sorted_tareas(j))-1
    end-if
end-do

qsort(true, s, sol_sorted)
writeln("\n\nOPTIMO GREEDY: ", opt)
writeln("\nTareas ejecutadas: ", sum(j in tareas|solucion(j)=1)solucion(j))
```

Figura 2.22: ISDP - Algoritmo *greedy*

(a) Código para obtener una solución con el algoritmo *greedy* en *Xpress*

´ Como se ha indicado en el apartado anterior, en primer lugar se **ordenan** las tareas por orden decreciente de beneficios.
Después, se toma el valor máximo de las *deadlines* y se establece como **disponible cualquier valor de 2 hasta ese máximo**, de forma que los valores de los tiempos de finalización, f_j, de las tareas puedan tomar cualquiera de esos valores. Para esto, se define inicialmente una lista de longitud max_d, el valor máximo de las *deadlines*, con valores inicializados a 1.

El siguiente paso es **recorrer la lista** de tareas, buscando una posición dentro de los valores de f_j disponibles que no supere el valor de la *deadline*, d_j, correspondiente.
Si el valor que devuelve la función ***buscar_intervalo*** para f_j es -1, significará que no existe un hueco para dicha tarea. En caso contrario, se añade dicha tarea a la solución y se suma su beneficio al valor óptimo. Se indica que ese valor de f_j ya no está disponible para el resto de tareas y se establece el valor de s_j a $f_j - 1$, ya que la duración de todas las tareas es unitario.

La función utilizada para buscar un valor de f_j factible, que se muestra en la

Figura 2.24, toma el valor de la *deadline* de dicha tarea, busca un valor desde 2 hasta el valor máximo de las *deadlines* que esté disponible y que sea menor o igual que la *deadline* de la tarea seleccionada y se va asignando a la variable max_f hasta que se deje de cumplir una de las condiciones, de forma que se quede con el mayor valor posible que pueda tomar f_j sin superar el valor de d_j y sin solaparse con otra tarea.

```
function buscar_intervalo(j: integer):integer
    deadline:=d(sorted_tareas(j))
    max_f:=-1
    forall(i in 1..max_d)do
        if(disponible(i)=1 and deadline>=i)then
            max_f:=i
        end-if
    end-do
    returned:=max_f
end-function
```

Figura 2.24: ISDP - Función *buscar_intervalo*

A continuación, se comprueba si existe alguna solución mejor con **búsqueda local**, en concreto realizando un intercambio entre una tarea que esté ya en la solución y otra que no, como se muestra en el código de la **Figura 2.25**.

```
final:=0
while(final=0)do
    mejora_max:=-999
    ! Intercambiar dos tareas
    forall(indi in 2..max_d)disponible_bl(indi):=disponible(indi)
    forall(j,k in 1..n|solucion(j)=1 and solucion(k)=0)do
        xl(j):=0
        indice1:=f(j)
        disponible_bl(indice1):=1
        xl(k):=1
        forall(i in 1..n|i<>j and i<>k)xl(i):=solucion(i)
        indice2:=buscar_hueco(k)
        ! Si no hay disponibilidad para la tarea k se mantiene la disponibilidad de la solucion inicial
        if(indice2=-1)then
            disponible_bl(indice1):=0
        ! Si hay disponibilidad se comprueba si existe una mejora
        else
            nuevo_objetivo:=calcular_objetivo(xl)
            antiguo_objetivo:=sum(s in tareas)b(s)*solucion(s)
            mejora:=nuevo_objetivo - antiguo_objetivo
            ! Si hay mejora se deja la disponiblidad como está
            if(mejora > mejora_max)then
                mejora_max:= mejora
                jmax:=j
                kmax:=k
                disponible_bl(indice2):=0
                valor_fmax:=indice2
                writeln("Mejora max ", mejora_max)
            ! Si no hay mejora se mantiene la disponibilidad
            else
                disponible_bl(indice1):=0
                f(j):=indice1
            end-if
        end-if
    end-do
    if(mejora_max<=0)then
        final:=1
    else
        solucion(jmax):=0
        solucion(kmax):=1
        f(jmax):=-1
        s(jmax):=0
        f(kmax):=valor_fmax
        s(kmax):=f(kmax)-1
        forall(ind2 in 2..max_d)disponible(ind2):=disponible_bl(ind2)
        writeln("Tarea ",jmax," con ", f(jmax), " --> tarea ", kmax, " con ", f(kmax))
        writeln("\n")
    end-if
end-do
```

Figura 2.25: ISDP - Búsqueda local

Al igual que para el *Weighted Interval Scheduling Problem*, se inicializa una variable *flag*, $final$, que tomará el valor 1 cuando deba terminar la ejecución del algoritmo. En cada iteración del bucle se declara la variable $mejora\_max$ con valor suficientemente pequeño y se establece en la variable *disponible_bl* la misma disponibilidad de tiempos de finalización de la solución actual.

Dentro del bucle, se busca una tarea dentro de la solución actual y otra externa, asignando respectivamente al índice correspondiente del *array* $x1$, el valor 0 y el valor 1, de forma que cambien respecto a cómo aparecen en el *array solucion*. El resto de valores del *array* $x1$ toman el mismo valor que en *solucion*.
Se busca el valor más cercano a la *deadline* de la tarea k-ésima y se asigna a la variable *indice*2. La función *buscar_hueco* es equivalente a la función *buscar_intervalo*, salvo porque consulta el *array disponible_bl*, en lugar del *array disponible*, para no modificar este último hasta dar con una solución mejor que la actual.
Si esta variable toma un valor distinto a -1 se calcula el valor del objetivo como suma de los beneficios de las tareas ejecutadas en caso de que se realizase el intercambio y se compara con el objetivo actual. Si la mejora obtenida es mayor que la mejora máxima hasta el momento, se seleccionan los índices que hayan generado este valor y también el valor de f_k correspondiente. En caso de que no se genere una mejora o simplemente no exista ningún valor de f_k asignable, se mantiene la disponibilidad inicial y no se modifican los tiempos de finalización.

Finalmente, si la mejora es inferior o igual a cero, se termina la ejecución y en caso de que sea mayor, se establece la solución que ha generado dicha mejora, eliminando el índice $jmax$ de la solución y agregando el de $kmax$ al tiempo que se modifican los valores de f_{kmax} y s_{kmax}, así como los valores del *array disponible*.

Soluciones obtenidas A continuación se muestran los valores obtenidos para cada fichero de datos con la formulación de programación entera en *Xpress*, el algoritmo *greedy* y tras aplicar la búsqueda local con 2 intercambios:

Fichero de datos	Nº tareas	Sol. *Xpress*	Tareas	Sol. *greedy*	BL 2 ints	Tareas
$deadprofit\_10.txt$	10	64	10	64	64	10
$deadprofit\_40.txt$	40	186	28	186	186	28
$deadprofit\_60.txt$	60	200	29	221	221	29
$deadprofit\_80.txt$	80	208	28	242	242	29
$deadprofit\_100.txt$	100	220	29	248	248	29
$deadprofit\_120.txt$	120	123	22	253	253	29
$deadprofit\_140.txt$	140	113	20	265	265	29
$deadprofit\_160.txt$	160	113	17	260	260	29
$deadprofit\_180.txt$	180	83	19	272	272	29
$deadprofit\_200.txt$	200	88	17	273	273	29

La obtención de soluciones con la formulación de programación entera en *Xpress* se extiende bastante en el tiempo a partir del fichero *deadprofit_60.txt*, por lo que, en ese y los siguientes ficheros, se detiene la ejecución del programa antes de que alcance el óptimo.
Se observa que en todos los casos la solución obtenida con el algoritmo *greedy* es igual o mejor al valor obtenido con *Xpress* y que además esta se obtiene en un tiempo mucho menor, aunque realmente no es seguro que el valor obtenido con este algoritmo sea el óptimo, queda claro que se obtiene una solución factible y aceptable en un tiempo más razonable que el necesario para obtener soluciones con *Xpress*.
En ningún caso la búsqueda local encuentra una solución mejor, ya que todos los valores disponibles de f_j son utilizados en la solución obtenida con el algoritmo *greedy*, dado que el valor máximo de la *deadline* para los ficheros de datos indicados es 30 y en la mayoría de casos se ejecuta el total de tareas contenidas en el fichero (10) o el máximo posible, es decir, 29.
Dado que el algoritmo *greedy* ordena inicialmente las tareas por orden decreciente de beneficios es imposible que un intercambio suponga un incremento del beneficio total.

El único caso en que la solución no da a entender que una mejora es imposible a simple vista es el del fichero *deadprofit_40.txt*. Sin embargo, al ver la solución más de cerca, puede verse que el único intervalo libre es el de tiempo de inicio 28 y tiempo de finalización 29 y la búsqueda local no permite situar una tarea ahí y mejorar el objetivo.

2.4. *Interval Scheduling with deadlines and release times*

En este problema se parte de los siguientes **datos iniciales**: N tareas con *deadlines* d_j y tiempos de salida r_j. El tiempo de salida de una tarea j es el momento r_j a partir del cual su tiempo de inicio, s_j puede tomar un valor.
Por otro lado, todas las tareas tienen el mismo tiempo de procesamiento p y serán ordenadas en orden creciente de sus *deadlines*.
Se supone que $d_j \geq r_j + p$ para todo j y que además el valor mínimo de r_j es 1.
El valor máximo de las *deadlines* se conocerá como d_{max}.
Todas las tareas que sean ejecutadas lo deben hacer por tanto en el intervalo [1, d_{max}].

El **objetivo** es encontrar un *schedule* del conjunto de tareas dado que maximice el número de tareas ejecutado.
Un ***schedule*** S es una función que asigna tiempos de inicio s_j a las tareas, de forma que las tareas sean ejecutadas entre sus tiempos de salida y sus *deadlines* y que no se solapen entre sí.
Se dice que un *schedule* es **canónico** si tiene las siguientes dos propiedades:

1. S está **desplazado a la izquierda** en el sentido de que: para cualquier tarea $j \in S$, el tiempo de inicio de j es, bien r_j o bien el tiempo de finalización de la tarea que precede a j.

2. Si $i, j \in S$ e i se ejecuta antes que j, entonces o bien i acaba antes de r_j o bien $d_i < d_j$.

Es fácil ver que cada *schedule* puede convertirse en uno canónico sin cambiar el conjunto de tareas completadas. Por ejemplo, si dos tareas $i, j \in S$ no cumplen la segunda condición, es decir, si $d_i > d_j$, pero i es ejecutada en r_j o después y antes de j, se puede intercambiar ambas tareas. Este proceso se realiza hasta conseguir un *schedule* que satisfaga la **segunda condición**.

Para cumplir la **primera condición**, se modifica el *schedule* de izquierda a derecha cambiando todas las tareas a la izquierda, bien a sus tiempos de salida, o bien al tiempo de finalización de la tarea anterior, dependiendo de cual sea el mayor.

Se conoce como ***makespan***, $C(S)$ al tiempo más tardío de finalización de una tarea en S. Si $m \notin S$ y $d_m \geq C(S) + p$, entonces se conoce como $S \oplus m$ o extensión de S al *schedule* obtenido a partir de S y la ejecución de la tarea m en tiempo de inicio $max\{C(S), r_m\}$.

2.4.1. Métodos para buscar soluciones

El **algoritmo de Carlier**, **[19]**, se basa en una aproximación de programación dinámica que procesa las tareas de izquierda a derecha, en orden de los tiempos de salida. Para cada tiempo t, construye un ***schedule* parcial** con las tareas que terminan su ejecución antes o en el momento exacto t.
También considera una cierta jerarquía entre *schedules* parciales. De forma intuitiva, un *schedule* parcial está por encima de otro si este otro no puede extenderse para producir un *schedule* mejor, independientemente de las tareas que se vayan a realizar después de t. El *schedule* calculado para cada tiempo tiene la propiedad de que está por encima de todos los *schedules* parciales hasta ese momento.

Concretamente, Carlier presenta dos algoritmos, uno para determinar la factibilidad de un *schedule* para tareas de igual duración y otro para maximizar la cantidad de las mismas que van a ser ejecutadas.

El **algoritmo de factibilidad** calcula para cada tiempo $x = 0, 1, ..., d_{max}$ un *schedule* S_x de tareas que serán ejecutadas en el intervalo [1, x]. Se dice que S_x es un ***schedule* activo** si contiene todas las tareas cuyas *deadlines* sean como mucho $C(S_x)$.
Los *schedules* S_x se calculan incrementalmente, de izquierda a derecha como se indica a continuación: En el momento x, se considera el conjunto H de tareas de forma que para cada tarea j contenida en el mismo, $r_j \leq x - p$, es decir, puede procesarse antes de x, sin contar las tareas contenidas en el conjunto S_{x-p}. Es decir, las tareas que salen antes o en el momento $x - p$ y no se han ejecutado en el *schedule* S_{x-p}. Si $H = \emptyset$, se toma $S_x = S_{x-p}$. En caso contrario, se elige la tarea con la *deadline* más temprana $m \in H$. Si $S_{x-p} \oplus m$ es un *schedule* activo, se toma $S_x = S_{x-p} \oplus m$, en caso contrario $S_x = S_{x-1}$. Tras calcular $S_{d_{max}}$, si este es igual al conjunto formado por todas las tareas, se obtiene un resultado satisfactorio. En caso contrario, el conjunto de tareas no es factible.

Para justificar la corrección de este algoritmo se considera la relación de jerarquía de los *schedules* parciales explicada antes. Bajo la suposición de que el conjunto global de tareas sea factible, en cada paso x, el *schedule* S_x está por encima del resto en el intervalo [1,x]. En particular, esto implica que, para $x = d_{max}$, el *schedule* resultante $S_{d_{max}}$ contendrá todas las tareas posibles. Este algoritmo, así como el de maximización no tiene en cuenta los valores de las *deadlines* de las tareas que no han terminado a la hora de elegir la siguiente tarea, solo su orden.

El **algoritmo de maximización** presenta el siguiente **pseudocódigo**:

${S_x}^0 = \emptyset$ para todo x y ${S_x}^k$ indefinido para $k \geq 1$
for $k \in 1, 2, ..., N$ **do**
  for $x = p + 1, p + 2, ..., d_{max}$ **do**

$H = j$ - S_{x-p}^{k-1} tal que $r_j + p \leq x$
$H' = \{\ j \in H : d_j \geq x\ \}$
if $H' = \emptyset$ **then**
$S_x{}^k = S_{x-1}^k$
else
m = tarea con d_m más temprano en H'
$S_x{}^k = S_{x-p}^{k-1} \oplus m$
end if
end for
end for
return $S_{d_{max}}^k$ para el valor de k mayor, tal que $S_{d_{max}}{}^k$ esté definido.

2.5. *Discrete Interval Scheduling Problem*

Además de la variante más conocida del problema de*Interval Scheduling* se puede considerar la posibilidad de que el tiempo de inicio de una tarea j no presente un valor único s_j ya conocido, sino que pueda seleccionarse de un conjunto $S_j = \{s_{j1}, s_{j2},...,s_{jk}\}$ dado.
Este tipo de problema está relacionado con los conocidos como problemas de ***scheduling* con restricciones de tiempo**, al igual que el problema de la sección anterior, ya que en estos, las tareas presentan una fecha de salida r_j, una *deadline* d_j y un tiempo de procesamiento t_j. Si $d_j = r_j + t_j$ para cada tarea j, entonces se trata de un problema de *Interval Scheduling*. Cuando se asume que hay un intervalo para cada posible ejecución de la tarea j, entonces se puede modelar como un problema de *Interval Scheduling* discreto.
En este caso se genera el problema de tener múltiples posibilidades de ejecución de una misma tarea, dados los distintos valores potenciales de los tiempos de inicio.

Autores como Nakajima, Hakimi y Spieksma han buscado soluciones para este problema. Más concretamente, para el caso en que se considera una única máquina y donde el **objetivo** es maximizar el número de tareas ejecutado. Este problema es APX-hard (donde APX es la abreviación de *approximable* o abordable), un subconjunto de los problemas NP-hard, **[20]**[**21**]. Por su parte, los autores Berman y Dasgupta describen un algoritmo de aproximación combinatorio para obtener soluciones en el caso en que las tareas presentan pesos o prioridades.

2.6. *Minimize Maximum Lateness*

Un caso similar a aquel en que se conocen los beneficios y una *deadline* se da cuando se conoce su duración t_j, que no sería, por tanto, unitaria; pero no el beneficio que genera la tarea j-ésima, **[9][10][11][7]**.
Se supone que el tiempo de inicio de todas las tareas, s_j toma el valor 1 inicialmente, para simplificar el problema.
Por tanto, los **datos iniciales** son: un conjunto de tareas, N, con *deadline* d_j y duración t_j.
El **objetivo** es conseguir una planificación de las N tareas que minimice el retraso máximo o *maximum lateness*. Se definirá el tiempo de finalización de cada tarea como $f_j = s_j + t_j$, donde s_j será 1 para la primera tarea ejecutada y después tomará el valor de f_j para la última tarea ejecutada. El retraso de una tarea se define como $l_j = max(f_j - d_j, 0)$ y se considera que la tarea se ha retrasado si $f_j > d_j$, es decir, si $l_j > 0$.
Para calcular el máximo retraso del problema tenemos $L = \max l_j$ con j en [0,n-1].

La **formulación** en este caso, como se indica en el primer capítulo, es:

Minimizar z

Sujeto a: $s_j + t_j - d_j \leq z$ (9.1)

$s_j + t_j - s_i \leq My_{ij}, \quad \forall i < j$ (9.2)

$s_i + t_i - s_j \leq M(1 - y_{ij}),$

$x_j \in \{0,1\}\ j = 1, ..., n$ (9.3)

$y_{i,j} \in \{0,1\}\ i, j = 1, ..., n$ (9.4)

$s_j \geq 1\ j = 1, ..., n$ (9.5)

$z \geq 0$ (9.6)

2.6.1. Algoritmo *greedy*

De nuevo, se consideran diferentes posibilidades, **[10][11]**, a la hora de ordenar las tareas:

- **Menor tiempo de procesamiento primero:** Considerar las tareas en orden ascendente de tiempo de procesamiento t_j.
- ***Deadline* más temprana primero:** Considerar las tareas en orden ascendente de *deadline*, d_j.

- **Menor tiempo de tolerancia** Considerar las tareas en orden ascendente de tolerancia: $d_j - t_j$. Es decir, primero las tareas que menor diferencia tengan entre su *deadline* y su tiempo de procesamiento.

A continuación se describen una serie de contraejemplos para justificar la elección tomada [**22**]:
En el **primer caso** puede darse la situación, por ejemplo, de que una tarea j tenga tiempo de procesamiento $t_j = 1$ y *deadline*, $d_j = 100$ y otra tarea k tenga valores de $t_k = 10$ y $d_k = 10$. Si se ejecuta primero la tarea j, ya que su tiempo de procesamiento es menor, retrasará la ejecución de la tarea k-ésima, dado que esta tiene un valor de su *deadline* menor y necesita ejecutarse la primera para no retrasarse.
En el **tercer caso** podría encontrarse una tarea j con valores $t_j = 1$ y $d_j = 2$ y una tarea k con tiempos $t_k = 10$ y $d_k = 10$. El valor de la tolerancia para j sería de 1 y el de k sería 0, sin embargo si se ejecutase primero la tarea k el retraso de la tarea j-ésima sería de 9, mientras que si se ejecutase primero la tarea j, el retraso de la tarea k-ésima sería solo de una unidad.
Por tanto, en este caso, parece óptimo elegir la **segunda opción** y ordenar las tareas por **orden ascendente de *deadline***, ya que las otras dos opciones pueden obligar a ejecutar tareas de forma que otras con *deadline* menor ya no puedan ser ejecutadas por no hacerlo a tiempo.

El **pseudocódigo**, [**9**][**10**][**7**], para resolver este problema sería:

Ordenar N por d_j creciente
Inicializar el tiempo de finalización y el tiempo de inicio actuales / $f = s = 1$
for j in N **do**
    $s_j = f$
    $f_j = s_j + t_j$
    $f = f + t_j$
end for
return $[s_j, f_j]$

Es decir, se ordenan las tareas por el valor de d_j, de forma que el tiempo de finalización y el tiempo de inicio iniciales sean 1, y se recorre la lista de tareas ordenadas por *deadline*, asignando al tiempo de inicio de la tarea correspondiente el tiempo de finalización de la anterior.

Prueba

Para probar que el algoritmo es correcto, [**9**][**10**][**7**], necesitamos probar tres propiedades. Se considera el concepto de **inversión**, [**11**][**22**], como dos tareas j y j' en un *schedule* tal que la tarea j-ésima se ejecute antes de la tarea j'-ésima y el valor del *deadline* de la tarea j' sea menor que el de la tarea j, es decir $d_{j'} < d_j$:

- Cualquier par de *schedules* con tiempo de inactividad nulo y cero inversiones tienen el mismo valor de L (valor del máximo retraso de las N tareas).
- La solución *greedy* tiene un tiempo de inactividad nulo y cero inversiones.
- Hay una solución óptima con tiempo de inactividad nulo y cero inversiones.

Si estas tres propiedades son ciertas, se sabe que existe una solución óptima con tiempo de inactividad nulo y cero inversiones y que la solución *greedy* presenta dichas características. Dado que dos soluciones con estas características tienen el mismo retraso máximo, la solución *greedy* debe ser óptima, al tener el mismo retraso máximo que la solución óptima y por tanto minimizarlo.

La prueba de la **primera propiedad** establece que, sean dos soluciones S_0 y S_1 cualesquiera, ambas con tiempo de inactividad nulo y cero inversiones, ambas ejecutan las tareas de forma consecutiva y que para cualquier par de tareas tal que $d_j < d_{j'}$, la tarea j-ésima debe ejecutarse antes que la j'-ésima en ambas soluciones.
Esto supone que ambas soluciones deben ser idénticas salvo por aquellas tareas para las que el valor del *deadline* sea el mismo. Estas tareas podrían ejecutarse de forma consecutiva pero en distinto orden, ya que no afectaría al retraso máximo de la solución.
Para demostrar que efectivamente esto no afectaría al valor a minimizar, se consideran dos tareas con igual valor del *deadline*, k y k' tal que $d_k = d_{k'} = d'$, de forma que intercambiar ambas tareas no cambia el retraso máximo de ambas. El tiempo de finalización de la última tarea ejecutada será $t_f = f' + t_k + t'_k$ donde f' es el tiempo de finalización de la tarea previa a ambas. Por tanto, el tiempo de máximo retraso de estas tareas es $max(t_j - d', 0)$ para todo j, que tendrá el mismo valor independientemente del orden de ejecución de las tareas k y k'. Para extender esta prueba a un número arbitrario de tareas con el mismo *deadline*, el argumento es el mismo salvo porque $t_f = f' + \sum t_j$, $\forall j \in A$ de forma que los tiempos de *deadline* sean los mismos.

La prueba de la **segunda propiedad** se obtiene de la definición del algoritmo. Dado que cada tarea empieza cuando termina la anterior, la solución resultante tiene un tiempo de inactividad nulo necesariamente. Puesto que organiza las tareas por orden creciente de *deadline* no puede haber inversiones.

En cuanto a la prueba de la **tercera propiedad**, se parte de una solución con tiempo de inactividad en primer lugar y con inversiones en segundo lugar.
Eliminar el tiempo de inactividad no puede aumentar el máximo retraso de la solución. Esto decrementaría el tiempo de finalización de la tarea a la que se elimine el tiempo de inactividad. Si se tiene en cuenta la definición del retraso $l_j = max(f_j - d_j, 0)$ se puede entender que decrementar el valor de f_j solo puede reducir el de l_j.
En cuanto a eliminar una inversión de la solución, esto tampoco puede aumentar

el tiempo de retraso máximo. Sean las tareas correspondientes a la inversión p y $p+1$, de forma que $d_{p+1} < d_p$. Si se modifica el orden de las tareas, el retraso máximo tras el cambio, l' será menor o igual que el actual.
Para demostrar esto, se tiene que $l_p = max(f - t_{p+1} - d_p, 0)$ y $l_{p+1} = max(f - d_{p+1}, 0)$ y por tanto, el máximo retraso de la solución inicial es $l = max(f - t_{p+1} - d_p, f - d_{p+1}, 0)$. Y al cambiar el orden de las tareas, este retraso máximo sería de la forma: $l' = max(f - d_p, f - t_p - d_{p+1}, 0)$.
Dado que $f - t_p - d_{p+1} < f - d_{p+1}$, se puede asegurar que en caso de ser estos los valores máximos l' sería menor que l. Por tanto, solo queda demostrar que $f - d_p \leq l$ y que por tanto, $l' \leq l$ en cualquier caso. Para ello, se demuestra que $f - d_p \leq f - d_{p+1}$. Al estar f en ambos lados de la desigualdad, se puede expresar de la siguiente forma: $-d_p \leq -d_{p+1}$ o lo que es lo mismo: $d_{p+1} < d_p$, como ya se sabe por la definición de inversión. Al eliminar las inversiones se obtiene una solución nueva con cero inversiones que necesariamente debe ser óptima, al suponer la solución inicial con inversiones, ya óptima. Por tanto, la solución con tiempo de inactividad nulo y cero inversiones es óptima.

Finalmente, se puede probar que la solución *greedy* S es óptima de la siguiente forma:

- Dada una solución S' óptima sin tiempo de inactividad y con el menor número de inversiones.
- Entonces, si S' no tiene inversiones, S = S'.
- En caso de que S' tuviera alguna inversión, eliminarla supondría reducir el número de inversiones y no aumentar el valor del *maximum lateness*, lo que llevaría a una contradicción respecto a la definición de S', ya que el supuesto es que su número de inversiones ya es el mínimo posible para que la solución sea óptima.

Complejidad

1. La ordenación de las tareas por orden creciente de *deadline* puede realizarse con un algoritmo de ordención, dentro de los cuales los más eficientes presentan una complejidad de $\boldsymbol{O(n \cdot log(n))}$.
2. Inicializar el tiempo de finalización y el tiempo de inicio actuales tiene una complejidad de $\boldsymbol{O(1)}$.
3. Recorrer la lista de tareas supone una complejidad de $\boldsymbol{O(n)}$, mientras que las asignaciones realizadas dentro de este bucle presentan una complejidad de $\boldsymbol{O(1)}$.
   Por tanto, la complejidad del algoritmo [**9**], se reduce a la complejidad del algoritmo de ordenación: $\boldsymbol{O(n \cdot log(n))}$.

Implementación

Los **ficheros de datos** que se van a utilizar para probar la implementación del algoritmo *greedy* y la formulación de programación entera para este problema presentan la siguiente estructura: la primera línea contiene el número de tareas, n, que se va a leer. Las siguientes n líneas contienen un primer número entero: el índice asociado a la tarea, un segundo número con el valor de d_j y un tercer número que contiene el valor de t_j para el índice correspondiente.
Los ficheros para los que se mostrarán las soluciones son: *mmlat_10.txt*, *mmlat_20.txt*, *mmlat_40.txt*, *mmlat_60.txt*, *mmlat_80.txt*, *mmlat_100.txt*, *mmlat_120.txt*, *mmlat_140.txt*, *mmlat_160.txt*, *mmlat_180.txt* y *mmlat_200.txt*.

La implementación del problema de **programación entera** en *Xpress* se puede ver en la **Figura 2.26**.

```
! MODELO  DE PROGRAMACIÓN ENTERA: MML - Óptimo

M:= 9999

! x(j) en {0,1}
forall(j in tareas)x(j)is_binary
! y(i,j) en {0,1}
forall(i,j in tareas)y(i,j)is_binary
! z>= 0
res_z:= z>=0

! Retraso máximo
max_lateness:=z

! Retraso de tarea j menor o igual que retraso máximo
forall(j in tareas)res_1(j):=init(j) + t(j) - d(j)<= z

! Restricciones de solapamiento
forall(i,j in tareas | i<j)res_21(i,j):= init(j) + t(j) - init(i) <= M*y(i,j)
forall(i,j in tareas | i<j)res_22(i,j):= init(i) + t(i) - init(j) <= M*(1-y(i,j))

! s(j)>= 1
forall(j in tareas)res(j):=init(j)>=1

! setparam("XPRS_MAXTIME", -10)

! Minimizar retraso máximo
minimize(max_lateness)
```

Figura 2.26: MML - Problema de programación entera con *Xpress*

Al igual que para el problema de *Interval Scheduling with Dealines* se asigna un valor lo suficientemente grande a M, la cota superior. Después se declaran las variables binarias, x_j e $y_{i,j}$ y se indica que el valor de z deberá ser positivo.
Se asigna z al óptimo, en este caso $max\_lateness$, el retraso máximo. En cuanto a las restricciones, la primera indica que la diferencia del tiempo de finalización de cada tarea, f_j, que se corresponde en este caso con la suma $init_j + t_j$, respecto de la *deadline* d_j, no debe superar el valor de z, ya que en ese caso este no sería el retraso máximo.
El otro par de restricciones obliga a ejecutar las tareas de forma que no se solapen. Se indica además que el valor del tiempo de inicio, $init_j$, debe ser mayor o igual que 1.

Por último, se establece que el objetivo es minimizar la variable $max\_lateness$.

La implementación del **algoritmo *greedy*** se muestra en la **Figura 2.27**.

```
procedure algoritmo_greedy

    !Ordenar tareas por d(j) creciente
    qsort(true, d, sorted_tareas)
    writeln("\nTareas ordenadas por deadline")
    writeln("\nj\td\tt")
    writeln("------------------")
    forall(j in tareas)writeln(sorted_tareas(j),"\t",d(sorted_tareas(j)), "\t", t(sorted_tareas(j)))

    f_act:=1
    s_act:=1
    forall(j in tareas)do
        s(sorted_tareas(j)):= f_act
        f(sorted_tareas(j)):=s(sorted_tareas(j)) + t(sorted_tareas(j))
        f_act:= f_act + t(sorted_tareas(j))
        dif:= f(sorted_tareas(j)) - d(sorted_tareas(j))
        if(dif>0)then
            l(sorted_tareas(j)):=dif
        else
            l(sorted_tareas(j)):=0
        end-if
    end-do

    L_min:=max(j in tareas)l(j)
    writeln("\n\nOPTIMO GREEDY: ", L_min)
    qsort(true,l,sol_sorted)
    writeln("\nj\ts\tf\tl")
    writeln("------------------------")
    forall(j in tareas)writeln(sol_sorted(j),"\t",s(sol_sorted(j)),"\t", f(sol_sorted(j)), "\t", l(sol_sorted(j)))

end-procedure
```

Figura 2.27: MML - Algoritmo *greedy*

En primer lugar, se ordenan las tareas en orden creciente de sus *deadline*.
A continuación, se establece el valor inicial del tiempo de finalización actual y del tiempo de inicio actual, que en este caso es 1, ya que todavía no se ha establecido el valor de los tiempos para ninguna tarea.

Dentro del bucle, se itera sobre la lista de tareas, ordenadas por d_j y se establece como s_j el valor de f_act, que se corresponderá con el tiempo de finalización de la última tarea o con 1 en el caso inicial.
Se asigna a f_j el valor de la suma de s_j y la duración de la tarea correspondiente, t_j y este último valor se suma al de f_act.
En caso de que la diferencia entre f_j y d_j sea mayor que cero, habrá un retraso en la ejecución de la tarea y será el valor de l_j, en caso contrario este será cero.

Por último, se calcula el valor máximo de l_j para todas las tareas, que será el óptimo de la solución obtenida con el algoritmo *greedy*.

Soluciones obtenidas. Puesto que se sabe que el algoritmo *greedy* proporciona el óptimo, no se ha considerado necesario implementar otros métodos para mejorar sus resultados y únicamente se compran sus soluciones con las proporcionadas por el *solver* de *Xpress*.
Los resultados obtenidos para los ficheros indicados anteriormente son los siguientes:

Fichero de datos	Nº tareas	Sol. *Xpress*	Sol. *greedy*
mmlat_10.txt	10	0	0
mmlat_20.txt	20	39	39
mmlat_40.txt	40	97	97
mmlat_60.txt	60	153	153
mmlat_80.txt	80	197	197
mmlat_100.txt	100	266	255
mmlat_120.txt	120	314	314
mmlat_140.txt	140	418	401
mmlat_160.txt	160	455	440
mmlat_180.txt	180	526	507
mmlat_200.txt	200	576	560

Las soluciones demuestran que efectivamente el algoritmo *greedy* obtiene valores iguales o mejores que el *solver* de *Xpress*. Además, estos se obtienen, en todos los casos, en un tiempo mucho menor.
El motivo de que algunos resultados con *Xpress* sean peores es que, para determinados casos, es necesario detener la ejecución porque esta se extiende demasiado en el tiempo.

2.7. *Minimize Mean End Time*

A partir de un conjunto de N tareas con tiempo de procesamiento t_j y prioridad p_j se puede considerar el **objetivo** de minimizar el siguiente sumatorio, [**7**]:

$$\sum_{j=1}^{n} w_j * f_j$$

donde w_j es el valor inverso de p_j.

2.7.1. Métodos de obtener el óptimo

En este caso obtener el óptimo consiste únicamente en ordenar las tareas en orden creciente del valor de: t_j/w_j o lo que es lo mismo, de: t_j*p_j.
Ejecutando las tareas en dicho orden se consigue minimizar el valor del sumatorio anterior.

Implementación

A continuación se muestra la solución obtenida para los **ficheros de datos** siguientes: *mmet_10.txt, mmet_20.txt, mmet_40.txt, mmet_60.txt, mmet_80.txt, mmet_100.txt, mmet_120.txt, mmet_140.txt, mmet_160.txt, mmet_180.txt* y *mmet_200.txt*. La estructura de estos ficheros es similar a la del resto de ficheros utilizados en los problemas anteriores: la primera línea contiene el número de tareas a ejecutar, n y las siguientes n líneas presentan un primer número correspondiente al índice de la tarea, un segundo número correspondiente a la prioridad, p_j y un tercer número que se corresponde con la duración de la tarea t_j.

La **implementación** en *Xpress* puede verse en la **Figura 2.28**.

En primer lugar se calcula el producto de t_j y p_j y se ordenan las tareas de forma que se ejecuten primero aquellas con menor tiempo de procesamiento y menor prioridad o menor valor de $1/w_j$, ya que esto implicaría ejecutar aquellas tareas cuyo valor de w_j sea alto con tiempos de finalización pequeños, minimizando así el producto de w_j y f_j. Se priorizan además las tareas de tiempo de procesamiento menor, de forma que una tarea larga no pueda obligar a ejecutar muchas con tiempo de finalización mayor.

Una vez ordenadas las tareas, se inicializan los tiempos de inicio y finalización y se itera sobre la lista de tareas, asignando el tiempo de finalización actual al tiempo de inicio de la tarea correspondiente. El valor de f_j será la suma de s_j y t_j de la tarea j-ésima, sumándose por último el tiempo de procesamiento de dicha tarea al tiempo de finalización actual.

```
!Ordenar tareas por t(j)*p(j) creciente
forall(j in tareas)product(j):=t(j)*p(j)
qsort(true, product, sorted_tareas)
writeln("\nTareas ordenadas por deadline")
writeln("\nj\tp\tt")
writeln("-------------------")
forall(j in tareas)writeln(sorted_tareas(j),"\t",p(sorted_tareas(j)), "\t", t(sorted_tareas(j)))

f_act:=1
s_act:=1

forall(j in tareas)do
    s(sorted_tareas(j)):= f_act
    f(sorted_tareas(j)):=s(sorted_tareas(j)) + t(sorted_tareas(j))
    f_act:= f_act + t(sorted_tareas(j))
end-do

forall(j in tareas)w(j):=1/p(j)
opt:=sum(j in tareas)w(j)*f(j)
writeln("\n\nOPTIMO GREEDY: ", opt)
writeln("\nj\ts\tf\tp\tw")
writeln("-----------------------------------")
forall(j in tareas)writeln(j,"\t",s(j),"\t", f(j), "\t", p(j), "\t", w(j))
```

Figura 2.28: MMET - Algoritmo *greedy*

Finalmente, se calcula la suma de los productos $w_j * p_j$, que será la solución óptima al problema.

Soluciones obtenidas. Los resultados obtenidos para los ficheros indicados anteriormente son los siguientes:

Fichero de datos	Nº tareas	Sol. *greedy*
mmet_10.txt	10	23.12
mmet_20.txt	20	111.35
mmet_40.txt	40	369.63
mmet_60.txt	60	710.08
mmet_80.txt	80	1298.94
mmet_100.txt	100	2536.87
mmet_120.txt	120	3314.93
mmet_140.txt	140	4402.76
mmet_160.txt	160	5708.03
mmet_180.txt	180	7352.64
mmet_200.txt	200	8864.28

Capítulo 3

Tareas con más de un procesador.

3.1. Problemas básicos: Versiones generalizadas del *ISMP* y el *WISP*.

3.1.1. Caso con prioridades desconocidas.

Uno de estos problemas se conoce como ***Interval Scheduling con máquinas dadas***. En este caso el número de máquinas a utilizar se considera fijo y el objetivo es maximizar el número de tareas que puedan ser ejecutadas.
Esta variante puede resolverse usando una formulación de flujo de mínimo coste. Concretamente en el caso en que no hay prioridades, se puede utilizar un algoritmo *greedy* para encontrar el número máximo de tareas.

Si cada tarea puede ser llevada a cabo solo por un subconjunto dado de máquinas el problema pasa a ser NP-hard y Kroon [**4**] propone en este caso una serie de heurísticas y algoritmos exactos.
En el caso de que exista un intervalo de disponibilidad asociado a cada máquina, Brucker y Nordmann [**4**] describen un algoritmo de complejidad O(n^{m-1}) que maximiza el número de tareas ejecutadas.

3.1.2. Caso con prioridades conocidas.

Interval Scheduling con tareas requeridas

El **objetivo** de esta variante [**4**], es llevar a cabo todas las tareas encontrando el *schedule* de coste mínimo o beneficio máximo.
Los **datos iniciales** de este problema para N tareas dadas son los costes o beneficios generados por cada tarea dependiendo de en qué máquina sea ejecutada.

***Interval Scheduling* con disponibilidadaes de máquinas**

Las máquinas pueden diferir además por **disponibilidad** [**4**], de forma que para cada máquina i de las m que pueden utilizarse, existe un intervalo $[a_i, b_i)$ donde puede darse un coste por utilizar la máquina i-ésima. En concreto para esta variante, decidir si existe un *schedule* factible es un problema NP-completo.

Además, se considera la posibilidad de que un intervalo de disponibilidad y un coste c_i estén asociados a un tipo de máquina i al que puede pertenecer un subconjunto de máquinas dado, en lugar de una sola máquina.

Para este problema Bathia propone un algoritmo de aproximación, también para el caso en que se pretenda minimizar el número de máquinas usado (siendo el coste igual para los diferentes tipos de máquinas).

***Interval Scheduling* jerárquico**

Otra variante considera un orden dado para las máquinas [**4**] y además una máquina $m(j)$ máxima para cada tarea j tal que $1 \leq m(j) \leq m$ y de forma que la tarea pueda ser procesada por las máquinas 1,..., $m(j)$, pero no por las máquinas $m(j)+1$,...,m. Utilizar cada máquina tiene un coste c_i, de forma que el **objetivo** es minimizar el coste de ejecutar todas las tareas. En este caso las máquinas están siempre disponibles.

De nuevo, decidir si existe un *schedule* factible es un problema NP-completo. En caso de que las máquinas se dividan en tipos y haya un orden lineal para estos tipos, Bathia propone nuevamente un algoritmo de aproximación.
Cuando solo hay dos tipos de máquinas, Dondeti y Emmons y Huanh y Lloyd muestran que el problema resultante se puede resolver en un tiempo polinómico, mientras que para tres tipos de máquinas el problema vuelve a ser NP-completo.

3.2. *Interval Scheduling Maximizing Minimum Coverage*

En este caso, el **objetivo** es seleccionar un subconjunto de tareas para ser ejecutadas por un número dado de máquinas, k, de forma que el mínimo de las máquinas que están ocupadas en un tiempo dado, sea lo mayor posible. Es decir, se busca aprovechar al máximo las máquinas en todo momento.

Esta variante de *Interval Scheduling* con varios procesadores se trata en la referencia [**23**], motivado por el problema de *minimum error correction* que busca el mínimo número de correcciones que permitan un particionado consistente de un conjunto de lecturas en dos copias del cromosoma a partir del que fueron secuen-

ciadas.
Para entender este problema se debe conocer la secuenciación de alto rendimiento, una técnica desarrollada las últimas décadas que puede producir millones de fragmentos de ADN, conocidos como **lecturas**, desde posiciones aleatorias a través del genoma de un individuo. Un análisis bastante detallado de las lecturas se conoce como *haplotype phasing* y tiene también en cuenta el hecho de que en algunas especies, como la humana, cada cromosoma se presenta en dos copias, heredadas de cada uno de los padres. En este contexto, es también deseable asignar las variaciones genéticas a la copia del cromosoma en el que están presentes.
Volviendo al problema de *minimum error correction*, cuanto mayor sea el número de lecturas y lo más uniformemente distribuidas a través del genoma, lo más exacta será la solución en la práctica. Sin embargo, la complejidad de este algoritmo hace que solo sea factible para valores pequeños de k.

En término de *Interval Scheduling*, el problema se define de la siguiente forma:

Los **datos iniciales** son un conjunto de lecturas j, con intervalos asociados $[s_j, f_j)$, de forma que $0 \leq s_j < f_j < N$.
Dado un punto $p \in [s_j, f_j)$, se dice que el intervalo $[s_j,\ f_j)$ lo **cubre** y que si además $p \neq s_j$ entonces lo **cubre estrictamente.** Dado un conjunto de intervalos (o tareas, siguiendo la nomenclatura de este documento), S= $\{[s_j,\ f_j) : j \in 1, ..., n$ / $s_j < f_j\}$ y un punto p se define la **cobertura de *p*** como $cov_S(p) = |\ [s_j,\ f_j) \in S$ / $[s_j,\ f_j)$ cubre $p|$.
También se define como **máxima cobertura de S** maxcov(S) = $max_{p\in[0,N)}$ $cov_S(p)$. Del mismo modo, se define como **mínima cobertura de S** mincov(S) = $min_{p\in[0,N)}$ $cov_S(p)$.
El **objetivo** es encontrar un conjunto $S' \subseteq S$ de forma que maxcov(S') $\leq k$ y maximizando mincov(S').

Es decir, se añade la restricción de maximizar $min_{p\in[0,N)}$ $cov_S(p)$ al problema que busca un conjunto de tareas máximo y factible para un número k de máquinas dado.

3.2.1. Métodos para buscar el óptimo

Este problema se puede resolver en un tiempo de $\boldsymbol{O(n^2 \cdot log(k)/log(n))}$.

Para ello, se considera una segunda versión o variante del problema, con los mismos datos de entrada, además de un valor entero t.
El **objetivo** será decidir si existe un conjunto $S' \subseteq S$ de forma que maxcov(S') $\leq$ k y mincov(S') $\geq t$ y si es así, devolver S'.

Para reducirlo a un problema de máximo flujo se sigue la notación estándar de

flujo de redes:
Sea $s = min_{i\in\{1,...,n\}} s_i$ y $f = max_{i\in 1,...,n} f_i$. Se construye un grafo $G_{S,k,t}$ (con posibilidad de arcos paralelos) cuyos vértices correspondan al conjunto $\{s-1, f+1\} \cup \{s_i, f_i : i \in 1, ..., n\}$. El vértice $s-1$ será el único nodo fuente del grafo y el vértice $f+1$ el único nodo pozo.
Para cada dos números consecutivos $i, j \in V(G)$ (es decir, de forma que no haya un número p en $V(G)$ con $i < p < j$), se añade el arco (i, j) a $E(G)$, donde $V(G)$ representa el **conjunto de vértices** del grafo y $E(G)$ el **conjunto de arcos**. Estos últimos son los arcos que forman la *columna vertebral* del grafo y concretamente, los arcos $(s-1, s)$ y $(f, f+1)$ tienen capacidad k, mientras que el resto tienen capacidad $k-t$. Para cada intervalo $[s_i, f_i) \in S$, se añade a $E(G)$ el arco (s_i, f_i) con capacidad 1. Estos últimos son los **arcos *intervalo***.

A continuación se demuestra que calcular la solución del problema de máximo flujo para $G_{S,k,t}$ es equivalente a resolver la **segunda versión** del problema. Para ello, se parte del siguiente teorema:
La segunda variante del problema admite una solución sobre una instancia (S, k, t) si y solo si el máximo flujo en $G_{S,k,t}$ tiene valor k y su solución puede obtenerse de cualquier flujo máximo integral en $G_{S,k,t}$

La **prueba** parte de que un conjunto $S' \subseteq S$ sea una solución de una instancia (S, k, t) de la segunda variante del problema. Se construye un flujo φ en $G_{S,k,t}$ asignando $\varphi(s-1, s) = \varphi(f, f+1) = k$. Dado que la capacidad de estos dos arcos es k y que no hay otro arco que salga desde $s-1$ o entre a $f+1$, esto implicará que φ es un flujo máximo en $G_{S,k,t}$. Para cualquier arco de tipo *intervalo* (s_i, f_i), se fija $f(s_i, f_i)$ a 1 si y solo si el correspondiente intervalo $[s_i, f_i)$ pertenece a S'. Finalmente, sea (i, j) cualquier arco de la *columna vertebral* del grafo, diferente a $(s-1, s)$ y a $(f, f+1)$; dado que para todo $p \in [i, j)$, p está cubierto por el mismo número de intervalos en S', $t_{i,j}$ con $t_{i,j} \leq t$, se fija $f(i, j) = k - t_{i,j}$.

De esta forma, φ satisface las restricciones de capacidad de $G_{S,k,t}$. Queda por tanto, demostrar que **la propiedad de conservación de flujo se mantiene para cada vértice salvo para el fuente y el pozo**. Sea i tal vértice y sea (i, j) su arco de la *columna vertebral* saliente y (l, i) su arco entrante de la *columna vertebral*. El valor del flujo saliente desde i equivale a $k - t_{i,j}$ más el número de intervalos en S' con i como extremo izquierdo. Esto equivale a k menos el número de intervalos que cubren i estrictamente.
De forma similar, el valor del flujo entrante a i equivale a $k - t_{l,i}$ junto con el número de intervalos que llegan a i. Esto también equivale a k menos el número de intervalos que cubren estrictamente a i, demostrando por tanto que se cumple la propiedad de conservación del flujo para i.

Para la implicación inversa, sea φ un flujo máximo integral en $G_{S,k,t}$ de valor k (de forma que exista un flujo integral porque todas las capacidades sean ente-

ras). La solución $S' \subseteq S$ de la segunda variante del problema consiste en aquellos intervalos $[s_i, f_i)$ de forma que el arco del correspondiente intervalo (s_i, f_i) tiene valor de flujo 1. Sea (i, j) un arco de la *columna vertebral* arbitrario en $G_{S,k,t}$ que no afecte ni al nodo fuente ni al nodo pozo. Se necesita demostrar que todo $p \in [i, j)$ está cubierto por al menos t intervalos de S' y por, como mucho k intervalos de S'. El punto p está cubierto por como mucho k intervalos porque f tiene valor k y por al menos t intervalos porque la capacidad del arco (i, j) de la *columna vertebral* es $k - t$. Véase que $G_{S,k,t}$ tiene $O(n)$ vértices y arcos. Por tanto, la segunda variante del problema puede resolverse en un tiempo $O(n^2/log(n))$ resolviendo el problema de máximo flujo de $G_{S,k,t}$.

En base a esto, se puede resolver la **primera variante** del problema (o el problema de maximización) mediante una técnica de **búsqueda binaria**. Si se aplica esto a $t = 1$, $t = 2$, $t = 4$,..., hasta encontrar t^+ donde para $t^- = 2t^+$ la segunda variante del problema (o el problema de decisión) no tenga una solución. La búsqueda binaria sobre instancias del problema de decisión con $t \in [t^+..t^-]$ da la cobertura mínima $t = OPT$ de la solución óptima para el problema de maximización.
Por tanto, **el problema de maximización o la primera variante es soluble en tiempo $O(n^2 \cdot log(OPT)/log(n))$, donde $OPT \leq k$ es la cobertura mínima de la solución óptima.**

Existe un **algoritmo de máximo flujo** a medida para obtener un mejor tiempo de ejecución cuando $k = o(n/log(n))$ basado en el algoritmo de máximo flujo de Ford-Fulkerson, que permite conseguir que la variante de decisión del problema se resuelva en un tiempo $O(nk)$ y la variante de maximización en tiempo $O(nk \cdot log(OPT))$ donde $OPT \leq k$, es la mínima cobertura de la solución óptima.

Aun así, la mejor complejidad conseguida es todavía de orden prácticamente cuadrático para valores grandes de k, por lo que se busca un algoritmo de aproximación de complejidad $O(n \cdot log(n))$.
Es necesario primero, extender la definición de mínima cobertura a los intervalos, de forma que $mincov([s_i, f_i)) = min_{p \in [s_i, f_i)} cov(p)$. Cuando un intervalo tiene una cobertura mínima menor o igual que $\lfloor k/2 \rfloor$ se dice que dicho intervalo es *crucial* y en caso contrario, que es *reemplazable*.

A continuación se prueba que: **dada una entrada (S, k) para el problema de maximización de la cobertura mínima, y un punto p, si $cov_S(p) = k' > k$, hay al menos $k' - k$ intervalos que cubren p y que son reemplazables.**

Esto se prueba **por contradicción**. Se supone que hay $k' > k$ intervalos que cubren p y que todos ellos son *cruciales*. Para cada intervalo *crucial*, existe al menos un punto contenido en él que tiene una cobertura menor o igual que $\lfloor k/2 \rfloor$. Sean estos los puntos de *soporte*. No es posible que p sea un punto de *soporte*, así

que los que lo sean deben ser mayores o menores que p. Si se separan estos en aquellos que son mayores que p y aquellos que son menores, uno de dichos conjuntos debe tener al menos $\lceil (k+1)/2 \rceil$ intervalos.
Sea este conjunto el de puntos mayores que p. Si se inspecciona el intervalo que tenga el menor punto de *soporte*, es decir, aquel más cercano a p, este deberá estar cubierto por la totalidad de los $\lceil (k+1)/2 \rceil$ intervalos que tienen el punto de *soporte* en la misma dirección. Por tanto, la cobertura de dicho punto de *soporte* debe ser igual o mayor que $\lceil k'/2 \rceil > \lfloor k/2 \rfloor$, lo que contradice el hecho de que ese sea un intervalo *crucial*.

El **algoritmo** es el siguiente:
El conjunto original de intervalos S será el conjunto de candidatos actuales. Para cada intervalo, se calcula la cobertura mínima y máxima. Esto permite detectar intervalos *cruciales*, que no serán nunca eliminados del conjunto S.
Entonces, los intervalos delimitadores son atravesados de izquierda a derecha. Siempre que se encuentra un delimitador p con cobertura k' mayor que k, entonces deben eliminarse $k' - k$ intervalos que cubran p del conjunto de candidatos. Se sabe que de entre los k' intervalos que cubren p, hay al menos $k' - k$ que son reemplazables, por lo que pueden eliminarse sin repercusiones.
Para cada intervalo $[s_i, f_i)$ que sea eliminado, se debe actualizar la cobertura de todos los delimitadores contenidos en dicho intervalo. Es fácil darse cuenta de que la solución óptima para el problema de maximización está contenida por k, y dado que el algoritmo nunca elimina los intervalos *cruciales*, el ratio de aproximación es $\rho = k/(\lfloor k/2 \rfloor)$.

3.3. Optimización del número de procesadores, *Interval Partitioning.*

El problema de *Interval Scheduling* que en lugar de partir de un número de máquinas fijo, busca optimizar el número de las mismas, se conoce como *Interval Partitioning* [**10**][**11**].
Los **datos de entrada** de este problema son los siguientes: Se parte de un número de tareas n de las cuales se conoce el tiempo de inicio s_j y el tiempo de finalización f_j.
El **objetivo** es encontrar el número de máquinas o procesadores mínimo para realizar todas las tareas sin que estas sean incompatibles entre sí.

La **formulación** del modelo, como se indica en la **Sección 3.3** es la siguiente:

Minimizar $\sum_{i=1}^{m} y_i$

Sujeto a: $\sum_{i=1}^{m} x_{ij} = 1 \ \forall j = 1, ..., n$ (7.1)

$$x_{ij} \leq y_i \ \forall i = 1, ..., m, \ j = 1, ..., n \quad (7.2)$$

$$x_{ij} + x_{ik} \leq 1, \forall i = 1, ..., m \ \forall j, k \mid f_j > s_k \text{ y } f_k > s_j \quad (7.3)$$

$$x_{ij} \in \{0,1\} \ i = 1, ..., m, \ j = 1, ..., n, \quad (7.4)$$

$$y_i \in \{0,1\} \ i = 1, ..., m \quad (7.5)$$

Se conoce como **gráfico de intervalos** al gráfico que se genera al identificar cada tarea con un nodo y cada arista entre dos nodos con el solapamiento entre ambas tareas. Encontrar el mínimo número de gráficos de distintos colores que incluyan tareas no conectadas es la solución para este problema.

El tamaño de un subconjunto de tareas que se solapan por parejas es un límite inferior del número de máquinas necesario para procesar todas las tareas. Por otro lado, el tamaño máximo de una instancia del problema básico de *interval scheduling* es un límite inferior del número de máquinas necesario. Usando el **teorema de descomposición de cadenas de Dilworth**, se puede demostrar que este número de máquinas es suficiente para procesar todas las tareas.
Un algoritmo para obtener un *schedule* óptimo es descrito por Ford y Fulkerson, que especificaron la conocida como **regla de la escalera** basada en el teorema de Dilworth. Esta regla supone O(n^2) operaciones.
Gupta propone otro procedimiento que se ejecuta con una complejidad de *O(n · log(n))*.

3.3.1. Límite inferior de la solución óptima

Se conoce como **profundidad** [**10**][**11**], de un conjunto de intervalos abiertos al número máximo que contiene un tiempo dado cualquiera. Es decir, equivale al número máximo de máquinas que estén ejecutando una tarea al mismo tiempo.
El número de máquinas buscado en un problema de *Interval Partitioning* es siempre mayor o igual a esta profundidad.

Se puede calcular la profundidad *d*, [**11**] de la siguiente forma:

Ordenar las tareas por tiempo de inicio s_j creciente
for $j \in N$ **do**
  for i tal que $i < j$ de forma que i y j se solapen **do**

```
        No considerar la etiqueta de i para j
    end for
    if Existe una etiqueta {1, 2, ..., d} que se pueda considerar para j then
        Asignar la etiqueta a j
    else
        No asignar etiqueta a j
    end if
end for
```

Se asigna una etiqueta a todas las tareas y las que se solapen nunca presentan la misma etiqueta.

3.3.2. Métodos para buscar el óptimo.

El **algoritmo *greedy*** para resolver este problema ordena las tareas en orden creciente de tiempo de inicio, s_j y asigna la tarea a cualquier máquina en la que no tenga incompatibilidades.
El **pseudocódigo** [**11**], sería equivalente al utilizado para calcular la profundidad:

```
Ordenar las tareas por tiempo de inicio s_j creciente
m = 0, con m el número de máquinas utlilizadas
for j ∈ N do
    if Tarea j es compatible con alguna máquina k de las m activas then
        Asignar tarea j a máquina k
    else
        Añadir una nueva máquina m + 1
        Asignar tarea j a la máquina m + 1
        m = m + 1
    end if
end for
return m
```

La **complejidad** de este algoritmo es de $\boldsymbol{O(n \cdot log(n))}$.

Prueba

La prueba de que el algoritmo *greedy* devuelve siempre la solución óptima [**10**], se basa en lo siguiente:

- Sea un número de máquinas m asignadas por el algoritmo, la máquina m-ésima está activa porque se ha necesitado para una tarea j incompatible con el resto de tareas asignadas a las $m - 1$ máquinas ya activas.
- Todas las tareas de la máquina m terminan después de s_j.

- Dado que las tareas inicialmente se han ordenado por tiempo de inicio, todas las incompatibilidades generadas con la tarea j son causadas por tareas que empiezan antes o al menos en el momento s_j.
- Por tanto, hay m tareas que se solapan con la tarea j en el momento $s_j + \epsilon$.
- Es necesario usar m o más máquinas.

Implementación

En este caso se utilizan los mismos ficheros que en la **Sección 2.1**, es decir, siguen la siguiente estructura: la primera línea indica el número de tareas a ejecutar, n, las siguientes n líneas tienen un primer número que es el índice de la tarea, seguido del tiempo de inicio s_j y el tiempo de finalización f_j.

La **implementación del problema de programación entera** en *Xpress* se muestra en la **Figura 3.1.**

```
! MODELO DE PROGRAMACIÓN ENTERA: IPP - Optimo

! x(i,j) en {0,1}
! y(i) en {0,1}
forall(i in maquinas,j in tareas)x(i,j)is_binary
forall(i in maquinas)y(i)is_binary

! Minimizar suma de máquinas
optimo:=sum(i in maquinas)y(i)

! Si una tarea es ejecutada lo será solo en una máquina
forall(j in tareas)res1(j):=  sum(i in maquinas)x(i,j)=1

! Si una tarea se ejecuta en una máquina esta máquina deberá
! estar activa
forall(i in maquinas,j in tareas)res2(i,j):= x(i,j)<= y(i)

! Si dos tareas se solapan no podrán ejecutarse en la misma máquina
forall(i in maquinas,j,k in tareas|j<>k and f(j)>s(k) and f(k)>s(j))
res3(i,j,k):= x(i,j)+x(i,k)<=1


minimize(optimo)

exportprob(EP_MAX, "IPP_sol", optimo)

writeln("\n\n\nNumero de maquinas optimo: ", getobjval)
```

Figura 3.1: IPP - Problema de programación entera

En primer lugar, se declaran las variables $x_{i,j}$ e y_i como binarias. Después, se define el óptimo como la suma de las máquinas que van a ser utilizadas. Donde el conjunto de índices i en *maquinas*, está comprendido entre los valores 1 y m, siendo m un número suficientemente grande.

La primera restricción, establece que para cualquier tarea j la suma de las variables binarias $x_{i,j}$ en i será 1, es decir, cada tarea se ejecutará solo en una máquina.
La segunda restricción indica que si una máquina no es usada entonces ninguna tarea podrá ser asignada a esta.
La última establece la restricción de solapamiento como en el problema *ISMP*, pero incluyendo el doble subíndice.
Finalmente se indica que el valor óptimo es un mínimo. Es decir, que se busca el mínimo número de máquinas donde puedan ejecutarse todas las tareas.

En cuanto a la implementación del **algoritmo *greedy***, que se muestra en la **Figura 3.2**:

```
procedure algoritmo_greedy
    ! Ordenar tareas por s(j) creciente
    qsort(true, s, sorted_tareas)
    writeln("\n\nTareas ordenadas segun f(j): \n")
    writeln("\nj\ts\tf")
    writeln("------------------")
    forall(j in 1..n)writeln(sorted_tareas(j), "\t", s(sorted_tareas(j)), "\t", f(sorted_tareas(j)))
    writeln("\n\n")

    ! Nº maquinas optimo inicializado a 1
    opt_greedy:= 1

    ! Recorrer tareas
    forall(j in tareas)do
        ! Buscar una maquina compatible con la tarea j
        maq:= buscar_maquina(sorted_tareas(j))
        ! Si no hay una compatible aumentar el optimo
        if(maq=-1)then
                opt_greedy:=opt_greedy + 1
                sol_greedy(opt_greedy, sorted_tareas(j)):=1
                writeln("Tarea ", sorted_tareas(j), " a maquina ", opt_greedy)
        ! Si la hay asignarla
        else
                sol_greedy(maq,sorted_tareas(j)):=1
                writeln("Tarea ", sorted_tareas(j), " a maquina ", maq)
        end-if
    end-do

    writeln("\n\nOPTIMO GREEDY: ", opt_greedy)
    nm:=0
    forall(i in 1..opt_greedy)do
        nm:=nm+1
        writeln("\nTAREAS DE LA MAQUINA ",nm,":")
        writeln("\nj\ts\tf")
        writeln("------------------")
        forall(j in tareas|sol_greedy(nm,j)=1)writeln(j,"\t",s(j),"\t",f(j))
    end-do

end-procedure
```

Figura 3.2: IPP - Algoritmo *greedy*

Como se indica en el pseudocódigo, primero se ordenan las tareas por orden creciente de sus tiempos de inicio, s_j. Después se inicializa el valor del óptimo *greedy* a 1, ya que como mínimo se va a necesitar una máquina.

Tras estos pasos, se itera sobre las tareas y para cada una de ellas se busca una máquina en la que sea compatible con el resto de tareas asignadas a dicha máquina.

En caso de que el índice de la máquina obtenido sea -1, significará que no hay ninguna máquina activa disponible en la que la tarea pueda ser ejecutada sin solaparse con otra tarea. Por tanto, se añade una máquina al óptimo y se asigna la tarea a esta.
Si el índice obtenido es distinto, este será el correspondiente a una de las máquinas activas en la que la tarea puede ser ejecutada sin solaparse con otras. En ese caso, se asigna la tarea a dicha máquina.

Una vez asignadas todas las tareas, se obtiene el óptimo como el número de máquinas que se han ido activando a lo largo de la ejecución del algoritmo.

Las funciones utilizadas para buscar una máquina compatible con cada tarea se muestran en la **Figura 3.3**.

```
! Buscar máquina compatible para la tarea j
function buscar_maquina(j:integer):integer
    ! Inicializar a no compatible
    k:= -1
    ! Buscar dentro de las ya activas
    forall(maqs in 1..opt_greedy)do
        ! Comprobar si son compatibles
        if(compatibles(maqs,j)=true)then
            returned:=maqs ! Si son compatibles devolver el valor
            k:=0
            break
        end-if
    end-do
    if(k=-1)then
        returned:=k
    end-if
end-function

function compatibles(k:integer, j:integer):boolean
    ! Suponer que son compatibles
    fact:=true
    ! Buscar en el array de tareas alguna dentro de la maquina
    forall(i in tareas)do
        ! Si la maquina ejecuta esa tarea
        if(sol_greedy(k,i)=1)then
            ! Comprobar si se solapa con la tarea a añadir
            if(f(j)>s(i) and f(i)>s(j))then
                fact:= false
            end-if
        end-if
    end-do
    returned:=fact
end-function
```

Figura 3.3: IPP - Funciones *buscar_maquina* y *compatibles*

La primera función, ***buscar_maquina*** inicializa una variable $flag$ a -1 y busca en todas las máquinas activas una compatible con la tarea pasada como argumento de la función. Si encuentra alguna, la variable $flag$ pasa a tomar el valor 0 y se devuelve el indice de la máquina correspondiente. En caso contrario se devuelve -1.
La segunda función, ***compatibles***, utilizada en la anterior, se encarga de comprobar si una máquina y una tarea son compatibles. Así la variable $flag$, $fact$, se inicializa a *true* hasta encontrar alguna tarea asignada a la máquina que se solape

con la tarea pasada como argumento. Si no encuentra ninguna tarea incompatible con esta, significará que la tarea puede ser asignada a la máquina y que por tanto son compatibles.

Soluciones obtenidas. A continuación se muestran los valores obtenidos para cada fichero de datos con la formulación de programación entera y el algoritmo *greedy*:

Fichero de datos	Número de tareas	Óptimo	Solución *greedy*
interval_10.txt	10	5	5
interval_20.txt	20	7	7
interval_40.txt	40	14	14
interval_50.txt	50	7	7
interval_80.txt	80	9	9
interval_100.txt	100	23	23
interval_120.txt	120	33	33
interval_150.txt	150	36	36
interval_170.txt	170	39	39
interval_200.txt	200	54	54

Se comprueba, efectivamente, que el método implementado obtiene el valor óptimo conseguido también con el problema de programación entera con *Xpress*.

Conclusiones.

Con la globalización es cada vez mayor la necesidad de obtener resultados de forma inmediata y eso lleva a buscar la eficiencia a la hora de ejecutar cualquier tipo de tarea.
Además, la exigencia de los clientes aumenta al crecer también las posibilidades a la hora de obtener servicios. Por tanto, es cada vez más crucial contar con medios que permitan conocer el orden óptimo para realizar determinadas actividades cuando el tiempo es limitado.

En algunas situaciones, los métodos que aseguran soluciones óptimas necesitan más tiempo del aceptable para obtener resultados o requieren más recursos de los que se pueden facilitar, sobre todo para aquellos problemas que, o bien no son básicos o pasan a ser complejos por sus dimensiones, como se ha visto en el caso del problema de *Interval Scheduling with Deadlines and Profits*.
En esos casos, utilizar algoritmos como los implementados en este trabajo puede ser suficiente para obtener los objetivos deseados y ofrecer al cliente resultados competentes.

En los casos más básicos, como es el caso del *Interval Scheduling Maximization Problem* o el del *Weighted Interval Scheduling Problem*, se obtienen resultados óptimos con el *solver* de *Xpress* en tiempos aceptables.
Algo parecido ocurre con el problema de *Interval Partitioning* que, sin embargo, requiere conocer una cota superior del número de máquinas necesarias, para no prolongar demasiado la búsqueda del óptimo.

No obstante, para el resto de variantes implementadas, como el problema de *Interval Scheduling with Deadlines and Profits* o los problemas de *Minimize Maximum Lateness* o *Minimize Mean End Time*, los algoritmos *greedy* utilizados obtienen soluciones factibles y en los dos últimos casos óptimas, en poco tiempo y con una implementación bastante sencilla.

Bibliografía

[1] Wiki. Activity selection problem. https://en.wikipedia.org/wiki/Activity_selection_problem#Weighted_Activity_Selection_Problem, 2021. Accesed: 2021-05-28.

[2] Wayne. Dynamic programming. https://www.cs.princeton.edu/~wayne/cs423/lectures/dynamic-programming-4up.pdf, 2021. Accesed: 2021-05-28.

[3] Weighted interval scheduling. https://courses.cs.washington.edu/courses/cse521/13wi/slides/06dp-sched.pdf. Accesed: 2021-08-23.

[4] Frits C.R. Spieksma. Interval scheduling. WebPage, 2007. Accesed: 2021-07-10.

[5] Jesús Sáez. Programación de tareas sobre intervalos (interval scheduling). Apuntes Modelos de Investigación Operativa. Accesed: 2021-04-05.

[6] CPPS. Interval scheduling. cpps interval scheduling. https://cpps.bacsbd.org/notebook/view-note/interval-scheduling, 2017. Accesed: 2020-10-01.

[7] Arash Rafiey. Interval scheduling. http://cs.indstate.edu/~arash/algolec9.pdf, 2016. Accesed: 2021-06-25.

[8] Wiki. Interval scheduling maximization problem. https://en.wikipedia.org/wiki/Interval_scheduling#Interval_Scheduling_Maximization, 2017. Accesed: 2021-05-28.

[9] Hartloff. Cse 331: Introduction to algorithm analysis and design greedy algorithms. https://cse.buffalo.edu/~hartloff/CSE331-Summer2015/greedy.pdf. Accesed: 2021-05-10.

[10] Éva Tardos Kevin Wayne, Jon Kleinberg. Algorithm design: Greedy algorithms. https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pearson/04GreedyAlgorithms.pdf, 2005. Accesed: 2021-05-22.

[11] T. M. Murali. Greedy algorithms. https://courses.cs.vt.edu/~cs4104/murali/Fall09/lectures/lecture-07-greedy-scheduling.pdf, 2009. Accesed: 2021-06-10.

[12] Dariusz Kowalski. Greedy algorithms: Interval scheduling, comp 523: Advance algorithmic techniques. https://lonati.di.unimi.it/algo/0910/lab/kowalski6.pdf, 2004. Accesed: 2021-09-11.

[13] S. Saurabh. Interval scheduling - greedy algorithm (earliest deadline first scheduling). https://www.youtube.com/watch?v=6VGXp6RGGJc, 2014. Accesed: 2021-10-31.

[14] FICO. Xpress optimization help: qsort function. https://www.fico.com/fico-xpress-optimization/docs/latest/mosel/mosel_lang/dhtml/qsort.html. Accesed: 2021-10-01.

[15] GeeksforGeeks. Job sequencing problem. https://www.geeksforgeeks.org/job-sequencing-problem/, 2021. Accesed: 2021-05-28.

[16] Abdul Bari. 3.2 job sequencing with deadlines - greedy method. https://www.youtube.com/watch?v=zPtI8q9gvX8, 2014. Accesed: 2021-10-31.

[17] Sphere Online Judge. Msched - milk scheduling. https://www.spoj.com/problems/MSCHED/, 2017. Accesed: 2021-05-28.

[18] Sphere Online Judge. Mia14b - work scheduling. https://www.spoj.com/MIA14/problems/MIA14B/, 2017. Accesed: 2021-05-28.

[19] Wojciech Jawor Lukasz Kowalik Maciej Kurowski Marek Chrobak, Christoph Dürr. A note on scheduling equal-length jobs to maximize throughput. https://www.researchgate.net/publication/227295198_A_Note_on_Scheduling_Equal-Length_Jobs_to_Maximize_Throughput/link/5791de6908aed51475ac1645/download, 2004. Accesed: 2021-08-10.

[20] Wiki. Np (complexity). https://en.wikipedia.org/wiki/NP_(complexity), 2021. Accesed: 2021-10-27.

[21] Wiki. Apx. https://en.wikipedia.org/wiki/APX, 2021. Accesed: 2021-10-27.

[22] Madhavan Mukund. Design and analysis of algorithms: Greedy algorithms - minimizing lateness. https://www.youtube.com/watch?v=z2OXv1qaSFE, 2015. Accesed: 2021-10-31.

[23] Staneva V. I. Tomescu A. y Valenzuela D. Mäkinen, V. Interval scheduling maximizing minimum coverage. https://reader.elsevier.com/reader/sd/pii/S0166218X17301488?token=1582449C54894B57A9E0D476C6A7E73B4ED1FF3F80BE13F71DAFB255B402727D3E14D94DFE07BCFC9originRegion=eu-west-1&originCreation=20211031135855, 2015. Accesed: 2021-07-22.

[24] Christos H. Papadimitrion Frits C.R. Spieksma Antoon W.J.Kolen, Jan Karel Lenstra. Interval scheduling: A survey. https://onlinelibrary.wiley.com/doi/epdf/10.1002/nav.20231, 2007. Accesed: 2021-07-10.

[25] Miguel Miní. Introducción al problema del máximo flujo. https://peon-pasado.github.io/competitive-programming-classes/web/class-22.html, 2020. Accesed: 2021-09-15.

[26] FICO. Xpress optimization help: floor function. https://www.fico.com/fico-xpress-optimization/docs/latest/mosel/mosel_lang/dhtml/floor.html. Accesed: 2021-10-04.

[27] FICO. Xpress optimization help: ceil function. https://www.fico.com/fico-xpress-optimization/docs/latest/mosel/mosel_lang/dhtml/ceil.html. Accesed: 2021-10-04.

[28] FICO. Xpress optimization help: round function. https://www.fico.com/fico-xpress-optimization/docs/latest/mosel/mosel_lang/dhtml/round.html. Accesed: 2021-10-04.

[29] FICO. Xpress optimization help: Arrays, sets, lists, records and user types. https://www.fico.com/fico-xpress-optimization/docs/latest/mosel/UG/dhtml/moselugB2.html. Accesed: 2021-10-02.

[30] FICO. Xpress optimization help: Arrays. https://www.fico.com/fico-xpress-optimization/docs/latest/mosel/UG/dhtml/moselugB2_sec_secB2initarr.html. Accesed: 2021-10-02.

[31] Mosel documentation: Modelling with xpress. https://www.maths.ed.ac.uk/hall/Xpress/XpressDocumentation.html. Accesed: 2021-10-07.

[32] FICO. Mosel language reference manual. https://www.fico.com/fico-xpress-optimization/docs/latest/mosel/mosel_lang/dhtml/GUID-D73D2F47-371D-3394-AC89-A299E0F7F291.html. Accesed: 2021-10-07.

[33] FICO. Xpress optimization help: Procedures and function. https://www.fico.com/fico-xpress-optimization/docs/latest/mosel/mosel_lang/dhtml/moselreflang_sec_seclangproc.html. Accesed: 2021-10-07.

[34] Fabian Hennecke Birgitta Weber Peter Widmayer Mark Cieliebak, Thomas Erlebach. Scheduling with release times and deadlines on minimum number of machines. https://link.springer.com/content/pdf/10.1007%2F1-4020-8141-3_18.pdf, 2001. Accesed: 2021-10-12.

[35] Jan Elffers. Scheduling with release times and deadlines. https://repository.tudelft.nl/islandora/object/uuid:5def2dbb-67d1-4672-a0b1-561d7dc1a74f/datastream/OBJ, 2014. Accesed: 2021-10-14.

[36] Sebastián Enrique Ortiz Gaitán. Modelo de programación lineal entera mixta para la programación de sistemas tipo *Job-Shop Flexible* en entornos *Make to Order*. https://dialnet.unirioja.es/servlet/articulo?codigo=6046313, 2017. Accesed: 2021-10-05.

[37] 4.2 scheduling to minimize maximum lateness. https://ocw.tudelft.nl/wp-content/uploads/Algoritmiek_Scheduling_to_Minimize_Maximum_Lateness.pdf. Accesed: 2021-10-31.