UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA INDUSTRIAL
DEPARTAMENTO DE INGENIERÍA MATEMÁTICA

PROGRAMACIÓN DE TRABAJOS EN LÍNEAS DE PRODUCCIÓN

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN GESTIÓN DE OPERACIONES
MEMORIA PARA OPTAR AL GRADO DE INGENIERO CIVIL MATEMÁTICO

FRANCO FABIÁN BASSO SOTZ

PROFESOR GUÍA:
JORGE AMAYA ARRIAGADA

MIEMBROS DE LA COMISIÓN:
RAÚL GOUET BARROS
ALEJANDRO JOFRÉ CÁCERES
CRISTIÁN CORTÉS CARRILLO

SANTIAGO DE CHILE
MARZO 2013

Resumen

En el presente trabajo se estudia el problema de envasado y embotellado de pedidos en líneas de producción. El problema es de tipo *scheduling* con características propias. La resolución del problema se aborda desde dos ángulos.
El primer enfoque consiste en plantear un problema de programación lineal mixto satisfaciendo las restricciones operacionales del sistema. Los resultados de esta primera estrategia satisfacen los requerimientos técnicos, sin embargo, los altos tiempos computacionales impiden su utilización para casos reales.
El segundo enfoque consiste en la utilización de un Algoritmo Glotón Usando *Constraint Programming* (AGUCP) más una estrategia de mejoramiento de la solución. AGUCP permite encontrar una solución factible al problema planteado en el modelo de programación lineal mixto con una calidad aceptable. En este caso, los tiempos computacionales son excelentes incluso para casos de gran tamaño. Sin embargo existe un porcentaje de entre el 15 % y el 20 % de los casos estudiados en los cuales el algoritmo no encuentra solución.
Se presenta además una mejora a la heurística AGUCP, la cual se denomina AGUCP++ y consiste básicamente en una implementación propia de AGUCP adaptando el modelo para enfocarse directamente en las variables de decisión de modo de insertarse mejor al espíritu del *Constraint Programming*. La implementación de este algoritmo fue hecha en *Python*. Las principales mejoras de este nuevo algoritmo son: (i) Se trabaja con una menor cantidad de variables debido al modo de guardar la información. (ii) El algoritmo entrega una solución, a pesar que, haya uno o más trabajos que no pudieron incorporarse. (iii) Se disminuye la cantidad de casos en los cuales no todos los trabajos son agendados a un 5 %. Esto depende esencialmente de cuan exigentes sea el caso de estudio. (iv) Los tiempos computacionales disminuyen en un 70 % en comparación con AGUCP

Finalmente, se incorpora una técnica de mejoramiento de la solución obtenida a través de AGUCP++, utilizando una estrategia basada en la técnica llamada *Local Search*. Estas búsquedas locales operan optimizando sobre un número acotado de trabajos -a partir de una solución inicial-, dejando fijos los demás. Esta estrategia permite, en poco tiempo, obtener mejoras sustantivas de la solución. Según los experimentos realizados, el porcentaje de mejora varía entre un 5 % y un 28 %.

Agradecimientos

A mis padres, Myriam y Patricio, por su apoyo y respaldo incondicional. A mis hermanos, Claudia, Leonardo y Cati por compartir sus experiencias profesionales y ayudarme en este proceso. A mis amigos personales Bernardo, Martin, Simón, Juan Antonio, Manuela por acompañarme en este camino. A mis compañeros de generación, y muy en especial a mis amigos del alma Mauricio, Tata y César por el estudio en conjunto y la amistad forjada. A mis socios en SolMat Francisco y Carlos por la ilusión del proyecto en el cual nos hemos embarcado. A Jorge Amaya por su preocupación y enseñanzas y a Francisca por su cariño y paciencia.

Tabla de Contenido

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

En muchas industrias productivas aparece el problema de planificación de la producción de productos en línea. Un ejemplo relevante en Chile es la industria vitivinícola, la cual está sometida a una fuerte competencia a nivel mundial. Una gran parte de la producción de vinos y derivados chilenos son enviados al exterior, representando una parte muy importante de las exportaciones del país. En ese sentido no sólo la calidad del producto importa, puesto que también los clientes exigen prontitud en la entrega y cumplimiento de los plazos.

Una parte muy relevante del proceso en esta industria, así como en otras similares (en general producción de productos alimenticios envasados) es la que corresponde al embotellado o envasado y a la fabricación de los lotes del producto para la entrega al cliente.

1.1. Definición de la problemática

En este trabajo abordaremos el problema en una perspectiva más general (no necesariamente ligada a una industria o empresa en particular). Eso quiere decir que plantearemos un modelo matemático que represente el problema del embotellado o envasado como un modelo decisional, en el cual el objetivo es ordenar en el tiempo una secuencia de trabajos elementales, que forman parte de una orden o pedido de un cliente. Por ejemplo, un cierto

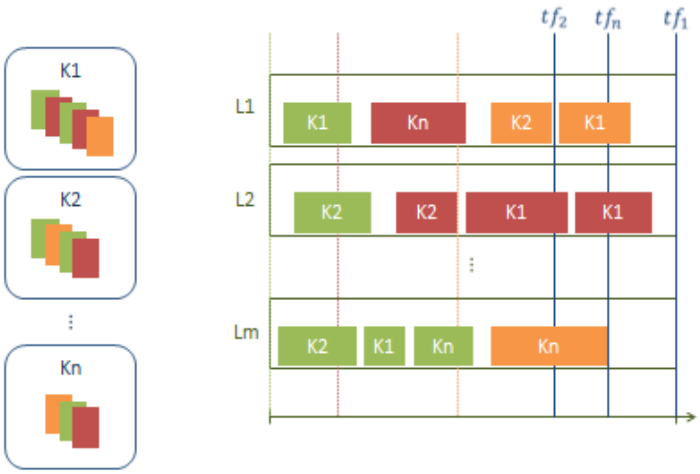

Figura 1.1: Esquema de solución al problema planteado

número de botellas de vino de ciertas cepas específicas puede corresponder a un trabajo. Un conjunto de trabajos distintos para un mismo cliente correspondería a una orden. Cada uno de estos trabajos está caracterizado por un vector de atributos, tales como tipo de producto (tipo de cepa, en el caso de los vinos), cantidad de unidades, tipo de envase o botella, tipo de corcho o tapa, fecha límite de despacho, duración del trabajo (la cual puede depender del equipo o linea de envasado en la cual se realiza), etc.

También consideraremos en el modelo la noción de línea de producción, que corresponde al equipo utilizado para realizar el trabajo (habitualmente, es una línea o correa sin fin, por donde circulan las unidades a envasar). Como normalmente hay varias líneas disponibles, también la asignación de un trabajo a una línea específica forma parte de la decisión del modelo. Por otra parte, la ubicación de un trabajo en el tiempo (secuencia) no es indiferente para el cumplimiento del objetivo de minimizar los tiempos de entrega. Es decir, entre dos trabajos sucesivos en general hay un tiempo dedicado a una proceso denominado *set-up*, que corresponde a las actividades necesarias para preparar la maquinaria para el trabajo i, cuando justo antes se ha realizado el trabajo j. De manera más precisa, en el caso del envasado de vinos, se necesita lavar los ductos o mangueras por donde ha circulado el producto, de manera de evitar la contaminación entre productos alterando su pureza. Esto puede ocurrir con cualquier producto alimenticio en proceso de envasado. Los tiempos de *set-up* son entonces parte relevante de los datos de entrada. Los valores de estos tiempos de *set up* no son simétricos, es decir no lo mismo pasar del trabajo i al trabajo j que pasar

del trabajo j al trabajo i. Este hecho deberá ser tomado en cuenta para tener un manejo adecuado de los datos en el programa.

La función objetivo estará generalmente ligada a la minimización del tiempo total de cumplimiento de las tareas encomendadas.

1.2. Nociones básicas

Gran parte del vocabulario que se definirá a continuación proviene del lenguaje vitivinícola, sin embargo es lo suficientemente amplio para cubrir un gran número de industrias, en especial con productos perecibles:

- Producto: Distintos SKU's [1] producidos por la empresa (por ejemplo vino rosado, merlot, blanco)
- Trabajo: Elemento básico que debe ser programado en las líneas de producción. Tiene asociado un tipo de producto y un volumen. Se considera como elemento indivisible, esto es, sólo puede realizarse completo o bien no realizarse.
- Orden: Pedido realizado por un cliente. Está compuesto por trabajos. Tiene asociado un fecha límite o *deadline* en el cual todos los trabajos de la orden deben haber sido procesados.
- Entrega: Corresponde a los insumos disponibles para realizar los trabajos. Tiene asociado una fecha inicial, fecha final o de caducidad y un volumen.
- Linea: Máquina procesadora de trabajo.

1.3. Ejemplo de aplicación: La industria vitivinícola

Una de las industrias que se adapta a la problemática de esta tesis es la industria vitivinícola. En efecto, el proceso de embotellado de vinos cumple con las siguientes cualidades requeridas por el modelo:

[1] Sigla en inglés para *Stock Keeping Unit*. Este término es empleado usualmente en lógistica para identificar los diferentes artículos o productos en el área de almecenamiento

- Producción en línea.
- Tiempos de *set up* para las máquinas.
- Tiempos máximo de despacho de pedidos.
- Insumos perecibles -dados por el área de enología-.

Además por su alto impacto en el sector exportador chileno parece un ejemplo interesante de desarrollar más en profundidad.

Figura 1.2: Linea de envasado en una viña

A continuación describiremos las distintas etapas en el proceso de embotellado del vino [2] :

Lavado y enjuagado de botellas

Los componentes de una línea de embotellado más frecuentes en las bodegas son: Lavadora de botellas, llenadora, taponadora o encorchadora, capsuladora y etiquetadora.

[2]Información obtenida del sitio especializado en vinos www.urbinavinos.com

Además, en líneas de altas capacidades podría completarse con despaletizadora de botellas previa al lavado, y equipos de encajonado y paletizado de las cajas.

Las anteriores partes de la línea de embotellado pueden ser más o menos independientes, o estar integradas en una única máquina compacta

Llenado de botellas

En el llenado se dosifica el vino con un volumen exacto, de forma que quede el espacio vacío necesario para alojar el tapón más una cámara de aire que permita cierta dilatación. Resulta de gran importancia el cuidado de la higiene en el embotellado, ya que influirá en la conservación posterior del producto.

Actualmente existen en el mercado muchos tipos diferentes de llenadoras, fabricadas para los distintos formatos de envases y con una amplia gama de rendimientos. Debido al elevado precio de las embotelladoras, resulta muy importante a la hora de seleccionar el equipo evaluar la capacidad de embotellado y su versatilidad.

Las máquinas de embotellado, constan de un depósito para la dosificación, un carrusel con sistema de arrastre, boquillas de llenado provistas de válvulas de nivel y mecanismos de avance y sujeción de la botella. En general se pueden distinguir tres sistemas de llenado:

- Por gravedad o caída libre, sistema en el cual no existe hermeticidad entre el gollete de la botella y la boquilla. El llenado se efectúa según el principio de los vasos comunicantes hasta el nivel de líquido del depósito central o bien, por gravedad. Dentro de estas llenadoras las hay de sifones descendentes o de botellas ascendentes. Es un sistema lento y sencillo, que se adapta muy bien a pequeños rendimientos.

- Isobarométricas, cuando existe hermeticidad, pero no hay diferencia de presión entre el gollete y el depósito de alimentación. Esa presión uniforme puede ser la atmosférica o superior a la atmosférica. Estas llenadoras se adaptan bien al embotellado de vino espumosos, siempre que se trabaje a contrapresión -superior a la atmosférica.

- A vacío o sobrepresión, cuando existe hermeticidad y se crea una diferencia de presión entre el depósito, que está a la presión atmosférica y el gollete, que está a una presión inferior, apareciendo así un vacío. Permiten el embotellado en caliente.

Los sistemas de llenado en los que existe hermeticidad entre el gollete y la boquilla permiten el embotellado bajo atmósferas inertes de CO2 y N2.

Las líneas modernas de altas producciones suelen estar automatizadas, disponiendo además sistemas de control de nivel de llenado.

Figura 1.3: Viñedos

Taponado

El taponado consiste en la introducción del tapón en el cuello de la botella, de manera que quede cerrado de forma hermética, dejando una cámara de aire entre la superficie de líquido y la del tapón que permita cubrir las dilataciones del primero. Se debe asegurar la hermeticidad del cierre evitando pérdidas de líquido. En el vino, el taponado se realiza con corcho, sobre todo si el producto es de calidad.

La normalización del gollete de las botellas ha permitido homogeneizar diámetros de tapón y cápsulas, facilitando la automatización del proceso de cierre. El taponado debe asegurar hermeticidad respecto a líquidos y gases a la vez que permitir una fácil apertura para su consumo.

En el cierre de las botellas con tapón de corcho se dan dos fases:

- La compresión del tapón mediante mordazas que disminuyen el tamaño del corcho (hasta un diámetro de casi 1/4 del diámetro inicial) para que pueda entrar en el cuello de la botella sin romperse. Una vez dentro el corcho recupera su tamaño asegurando

la estanqueidad, ni perder la capacidad de recuperación. Las mordazas pueden tener varias configuraciones (triple, cuádruple, lateral, de rodillos, etc.).

- La penetración del tapón, avanzando mediante un pistón que lo empuja dentro de la botella deslizándose desde la mordaza. La carrera del pistón debe estar bien regulada para que el tapón quede justo en la superficie del gollete.

En ocasiones se somete a los tapones de corcho a una preparación previa, por razones higiénicas y por que las propiedades de corcho mejoran al estar húmedo, enjuagándolos con agua o esterilizándolos con vapor.

Figura 1.4: Embotellado del vino

Una taponadora de corchos está compuesta por los siguientes elementos:

- Mordazas de compresión
- Émbolo o pistón accionado mecánica o neumáticamente.
- Cono centrador, cuyo perfil se adapta al gollete.
- Sistema de centrado de la botella, que puede ser de estrella o de guías.
- Soporte y muelle de compresión, para adaptarse a diferentes alturas de botellas.

Capsulado o sobretaponado

En el capsulado se dispone un elemento denominado cápsula que cubre el tapón y el gollete de la botella, con el fin de asegurar la limpieza de estos y garantizar el contenido del recipiente contra llenados fraudulentos. También proporcionan una zona de personalización del embotellado. La cápsula debe quedar lisa y ajustada a la botella.

Las cápsulas de los vinos, principalmente de calidad, se han realizado tradicionalmente de plomo, pero por motivos sanitarios se ha prohibido la utilización de este material. Los principales tipos de cápsulas son los siguientes:

- Cápsulas embutidas monopieza (de estaño o aluminio).
- Cápsulas pegadas de dos piezas (de aluminio, de complejo estándar o de material termorretráctil).
- Cápsulas de plástico.

En el capsulado en primer lugar se distribuye la cápsula mediante aire comprimido o por medios mecánicos, para, posteriormente alisarse quedando ajustada al cuello de la botella. El alisado puede ser mecánico con rulinas de goma o muelle para cápsulas de estaño y aluminio, o bien por calor en cápsulas de material termorretráctil (en capucha que baja sobre la botella o en túnel que es atravesado por la botella).

Etiquetado

El etiquetado es la disposición de las etiquetas sobre la botella. Se debe realizar justo antes de la salida al mercado para evitar su deterioro. En la botella de vino se pueden colocar principalmente tres tipos de etiquetas: etiqueta frontal, contraetiqueta y etiqueta collarín. Las etiquetas pueden ser autoadhesivas o encoladas, estas últimas cada vez menos utilizadas.

Según el tipo de etiqueta a colocar se pueden encontrar en el mercado distintos modelos de etiquetadoras. Las botellas son transportadas sobre una cinta, entrando en un carrusel conducidas por guías o estrellas.

En etiquetas pegadas con cola, las etiquetas se frotan contra un cilindro porta-cola, alimentado por una bomba o por gravedad. A continuación, son recogidas por el cilindro de etiquetado, que mediante unas pinzas las sujeta las deposita contra el cuerpo de la botella. Este sistema se repite en el perímetro del carrusel para los distintos tipos de etiquetas. En

etiquetas autoadhesivas, éstas se encuentran en rollos que permiten mediante un sistema giratorio su disposición sobre la botella, unos rodillos de goma las extienden dejándolas lisas. Es muy importante el correcto alineado de las etiquetas.

Capítulo 2

Revisión bibliográfica

El problema al cual se refiere esta tesis tiene ciertas similitudes con el renombrado problema llamado *Flowshop-scheduling problem* el cual puede explicarse, brevemente, como sigue: consideramos un grupo de m máquinas y un conjunto de n trabajos, los cuales deben ser procesados en estas máquinas. Cada uno de los n trabajos debe ir pasando a través de las m máquinas en un orden predeterminado. Cada trabajo puede ser procesado por una y una sola máquina a la vez. Es decir, consideramos los trabajos como indivisibles. Se consideran además tiempos de *set up* independientes de la secuencia. El objetivo del problema es especificar orden e instante para el procesamiento de los trabajos en las máquinas optimizando alguna función de los tiempos (por ejemplo tiempo final de ejecución del último trabajo usualmente llamado *makespan*).

Uno de los primeros investigadores en analizar propiedades de este tipo de problemas fue Johnson (1954) [7], sin embargo la notación clásica usado hasta nuestros dias para este problema, se debe a Conway et al (1967) [5]. Los primeros intentos de dar solución a este problema utilizaban procedimientos exactos como branch and bound -Lomnicki (1965) [11]-, o dynamic programming -Held et al (1962) [6]-. Para este tipo de problemas existen, en general $n!^m$, diferentes maneras de secuenciamiento por lo que heurísticas han debido ser usadas. Por ejemplo Campbell et al (1970) [3] propone un algoritmo simple utilizado para grandes instancias capaz de resolver con bajo margen de error instancias de hasta 60 trabajos (considerando computadores de esa época).

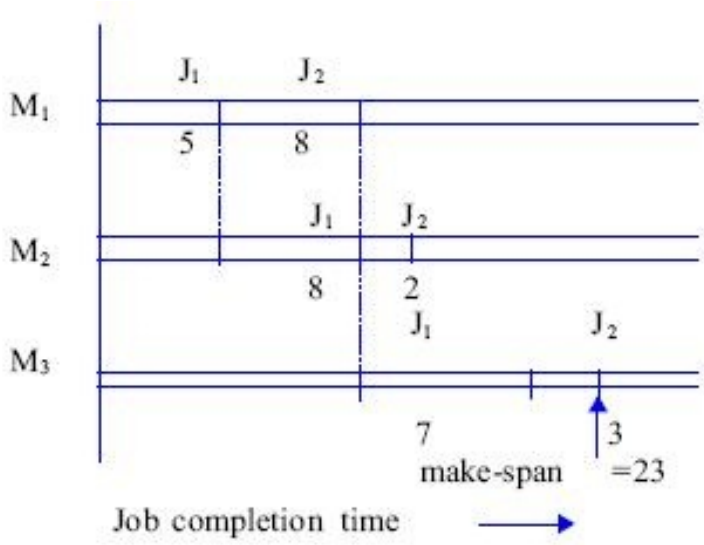

Figura 2.1: Ejemplo de solución para el *Flowshop-scheduling problem*

Existe una gran cantidad de variantes de este problema clásico. Por ejemplo Aldowaisan y Allahverdi (1998) [2] presentan una heurística para resolver el *Flowshop-scheduling problem* con la restriccón adicional que cada trabajo, una vez que inicia su procesamiento a través de las máquinas, no puede tener tiempos muertos, es decir tiempos en los cuales alguna máquina no lo esté procesando.
Por su parte Prabhakar (1974) [12] trata un problema levemente distinto, incluyendo producción al scheduling. Prabhakar considera cantidades continuas de material procesado en una máquina a cierta velocidad y luego yendo de una máquina a otra dentro de la cadena productiva. En ese sentido, el concepto de trabajo desaparece. La estrategia seguida por Prabhakar es el modelamiento de un *mixed integer programming* la cual debido al carácter combinatorial del problema resulta en resolución de pequeñas instancias solamente.

Las diferencias existentes entre los problemas más trabajados en la literatura y el problema planteado en esta tesis, se concentran alrededor de los siguientes puntos:

- Los trabajos se procesan por una y una sola máquina.
- Existen varias máquinas capaces de realizar el mismo trabajo, pero a velocidades distintas (líneas en paralelo).

- Ventanas de tiempo (denominadas entregas) para realización de los trabajos según el tipo de producto al cual pertenecen.
- Capacidades volumétricas en cada entrega.
- Tiempos de *setup* dependiente de los trabajos y de las líneas.
- Cota inferior para el inicio de cada trabajo, debido a la necesidad de insumos particulares -para el caso de los vinos: corchos, etiquetas, botellas etc-.
- Cota superior para el fin de cada trabajo dado por la orden -o cliente- al cual pertenece.

En resumen, el estado del arte presenta variados soluciones para problemas cercanos al presentado en esta tesis por lo que representaron un buen punto de partida para las idea que aquí se expondrán.

Capítulo 3

Un modelo de programación lineal mixto

3.1. Preliminares

La solución propuesta en este capítulo consiste en desarrollar un modelo de optimización lineal mixto que resuelva el problema. Para ello nos valdremos esencialmente de dos tipos de variables. En un primer lugar consideraremos variables binarias decisionales: se debe decidir a cuál línea de producción y a cuál entrega (insumo de vino por ejemplo) se asocia cada trabajo. El segundo tipo de variable corresponde a variables naturales: se debe decidir en qué instante se comienza a producir el trabajo. A continuación presentamos la formulación matemática del modelo.

3.2. Hipótesis de trabajo

- Se consideran varios tipos de productos.
- Los tiempos de setup sólo dependen del trabajo inicial, final y de la línea en que se hace el trabajo.
- Los tiempos para cada trabajo solo dependen de la línea.
- Se permite que varias entregas se asocien a un mismo trabajo sin que ello implique tiempo extra.

3.3. Conjuntos de interés

- K: Conjunto de órdenes.
- N: Conjunto de trabajos.
- P: Conjunto de productos.
- L: Conjunto de líneas.
- E: Conjunto de entregas.

3.4. Datos de entrada

3.4.1. Constantes

- M: Horizonte de tiempo.
- $setup_{n,n',l}$: Tiempo de setup necesario para pasar del trabajo n al trabajo n' en la línea l.
- $VOLMIN$: Mínimo volumen que puede ocuparse para una entrega en un trabajo.
- NG: Número suficientemente grande.

3.4.2. Trabajos

- $tin_{n,p} \in \{0,1\}$: Toma el valor 1 si el trabajo n es del producto p y 0 si no.
- $h_n \in \mathbb{N}$: Tiempo a partir del cual están disponibles los insumos para procesar el trabajo n.
- $rc_{l,n} \in \mathbb{N}$: Tiempo que se demoraría el trabajo n si es procesado en la línea l.
- $lt_n \in \mathbb{N}$: Cantidad de litros del trabajo n.

3.4.3. Órdenes

- $I_k \subseteq N$: Trabajos asociados a la orden k.
- $t_k \in \mathbb{N}$: Tiempo en el cual es requerido que esté lista la orden k.
- $w_k \geq 0$: Ponderación de importancia que se la da a la orden k.

3.4.4. Entregas

- $ti_e \in \mathbb{N}$: Tiempo inicial de la entrega e.
- $tf_e \in \mathbb{N}$: Tiempo final de la entrega e.
- $vol_e \in \mathbb{N}$: Cantidad de litros de la entrega e.
- $tinto_{e,p} \in \{0,1\}$: Toma el valor 1 ssi la entrega e es del producto p.
- $e_0 \in E$: Entrega artificial que indica que no se sabe cual entrega es (trabajos incorporados a priori).

3.5. Variables

- $g_n \in \mathbb{N}$: Tiempo en el que se empieza a procesar trabajo n.
- $x_{n,l} \in \{0,1\}$: Toma el valor 1 ssi el trabajo n se precesa en la línea l.
- $y_{n,n'} \in \{0,1\}$: Toma el valor 1 ssi el trabajo n se procesa después que el trabajo n' en la misma línea.
- $yj_{n,n'} \in \{0,1\}$: Toma el valor 1 ssi el trabajo n se procesa justo despues del trabajo n' en la misma linea
- $zb_{n,e} \in \{0,1\}$: Toma el valor 1 ssi el trabajo n se realiza con la entrega e.
- $z_{n,e} \in [0,1]$: Porcentaje de litros que aporta entrega e al trabajo n.
- $o_n \in \mathbb{N}$: Tiempo en el que se completa el trabajo n.
- $f_k \in \mathbb{N}$: Tiempo en el cual se han completado todos los trabajos de la orden k.
- $u_k \geq 0$: Parte positiva de $f_k - t_k$.

3.6. Restricciones

(1)	u_k	$\geq$	$f_k - t_k$	$\forall k \in K$
(2)	$\sum_{l \in L} x_{n,l}$	$=$	1	$\forall n \in N$
(3)	g_n	$\geq$	h_n	$\forall n \in N$
(4)	f_k	$\geq$	o_n	$\forall \in K, \forall n \in I_k$
(5)	o_n	$=$	$g_n + \sum_{l \in L} x_{n,l} \cdot rc_{n,l}$	$\forall n \in N$
(6)	$y_{n,n'}$	$\leq$	$1 + \frac{g_n - g_{n'}}{M+1}$	$\forall n, n' \in N, n \neq n'$
(7)	$y_{n,n'}$	$\leq$	$1 - x_{n,l} + x_{n',l}$	$\forall n, n' \in N, n \neq n', \forall l \in L$
(8)	$y_{n,n'}$	$\geq$	$x_{n,l} + x_{n',l} - 2 + \frac{g_n - g_{n'}}{M+1}$	$\forall n, n' \in N, n \neq n', \forall l \in L$
(9)	1	$\geq$	$x_{n,l} + x_{n',l} - y_{n,n'} - y_{n',n}$	$\forall n, n' \in N, n > n'. \forall l \in L$
(10)	$o_{n'}$	$\leq$	$2M(1 - yj_{n,n'}) + 2M(2 - x_{n,l} - x_{n',l}) + g_n - setup_{n',n,l}$	$\forall n, n' \in N, n \neq n', \forall l \in L$
(11)	g_n	$\leq$	M	$\forall n \in N$
(12)	$\sum_{e \in E} z_{n,e}$	$=$	1	$\forall n \in N$
(13)	$tin_{n,p} \cdot zb_{n,e}$	$\leq$	$tinto_{e,p}$	$\forall n \in N, \forall e \in E, \forall p \in P$
(14)	$(1 - tin_{n,p}) \cdot zb_{n,e}$	$\leq$	$1 - tinto_{e,p}$	$\forall n \in N, \forall e \in E, \forall p \in P$
(15)	o_n	$\leq$	$tf_e \cdot zb_{n,e} + M(1 - zb_{n,e})$	$\forall n \in N, \forall e \in E$
(16)	$ti_e \cdot zb_{n,e}$	$\leq$	g_n	$\forall n \in N, \forall e \in E$
(17)	$\sum_{n \in N} lt_n \cdot z_{n,e}$	$\leq$	vol_e	$\forall e \in E$
(18)	$z_{n,e}$	$\leq$	$zb_{n,e}$	$\forall n \in N, \forall e \in E$
(19)	$zb_{n,e}$	$\leq$	$NG \cdot z_{n,e}$	$\forall n \in N, \forall e \in E$
(20)	g_n	$\geq$	1	$\forall n \in N$

$$(21) \quad z_{n,e} \cdot lt_n + NG \cdot (1 - zb_{n,e}) \geq VOLMIN \qquad \forall n \in N, \forall e \in E$$

$$(22) \quad yj_{n',n} \leq y_{n',n} \qquad \forall n, n' \in N, n \neq n'$$

$$(23) \quad yj_{n,n'} \leq 2 - y_{n,n''} - y_{n'',n'} \qquad \forall n, n', n'' \in N, n \neq n', n \neq n'', n' \neq n''$$

$$(24) \quad y_{n,n'} \leq \sum_{n'' \in N} yj_{n,n''} \qquad \forall n, n' \in N, n \neq n'$$

$$(25) \quad yj_{n,n} = 0 \qquad \forall n \in N$$

Análisis de las restricciones

A continuación se analizarán las restricciones del modelo de modo de ver en qué medida incorporan las restricciones provenientes de la problemática expuesta en la definición de la problemática.

(1) Definición de u_k como la parte positiva de $f_k - t_k$. Además deberá incluirse u_k en la función objetivo para lograr el efecto deseado.

(2) Un trabajo se asigna a una y solo una línea.

(3) Para empezar a realizar un trabajo deben estar los insumos correspondientes.

(4) Definición de f_k como $\max_n o_n$. Además deberá incluirse f_k en la función objetivo para lograr el efecto deseado.

(5) Definición de o_n como el tiempo final de realización del trabajo n.

(6) Definición de $y_{n,n'}$. Se obliga que $y_{n,n'}$ valga cero en caso que el inicio del trabajo n' sea posterior a n.

(7) Definición de $y_{n,n'}$. Se obliga que $y_{n,n'}$ valga cero si n' y n no se procesan en la misma línea.

(8) Definición de $y_{n,n'}$. Se obliga que $y_{n,n'}$ valga uno si los trabajos se hacen en la misma línea con n posterior a n'.

(9) Definición de $y_{n,n'}$. Si ambos trabajos están en una misma línea entonces uno y sólo uno de los y vale 1.

(10) Restricción por tiempos de *set up*.

(11) Limite dado por el horizonte de planificación.

(12) Cada trabajo de ser completamente usando una o varias entregas.

(13) Una entrega se puede solo asociar a un trabajo del mismo producto.

(14) Una entrega se puede solo asociar a un trabajo del mismo producto.

(15) Si un trabajo se asocia a una entrega, el trabajo debe terminar antes del final de la entrega.

(16) Si un trabajo se asocia a una entrega, el trabajo debe empezar después del inicio de la entrega.

(17) La suma de los volúmenes de los trabajos asociados a una entrega no puede sobrepasar su capacidad.

(18) Definición de $zb_{n,e}$. Si el trabajo no utiliza insumo e entonces el porcentaje de utilización de e es cero.

(19) Definición de $zb_{n,e}$. Si se utiliza el insumo e entonces el porcentaje de utilzación del insumo e es estrictamente positivo.

(20) Cota inferior g_n.

(21) Utilizar una entrega en el procesamiento de un trabajo sólo puede hacerse si el volumen utilizado es mayor que un volumen mínimo.

(22) Definición de $yj_{n,n'}$. Si el trabajo n no se procesa después que el trabajo n' entonces tampoco se procesa justo después.

(23) Definición de $yj_{n,n'}$. Solo una de las variables $yj_{n,n'}$ e $yj_{n',n}$ vale uno.

(24) Definición de $yj_{n,n'}$. Para cada par de trabajos n y n' si no hay trabajo n'' que este justo después de n entonces $yj_{n,n'} = 0$.

(25) Definición de $yj_{n,n'}$. Un trabajo no se procesa justo después que sí mismo.

3.7. Función objetivo

$$\min \sum_{k \in K} w_k (f_k + u_k) + \frac{1}{|N|} \sum_{n \in N} o_n$$

La función esta compuesta por dos términos los cuales buscan un objetivo principal y secundario respectivamente. El primer término busca minimizar los retrasos en el despacho de las órdenes. Cabe recordar que w_k corresponde al peso que se le da a la orden k. w_k puede también considerarse como el costo por unidad de tiempo de no satisfacer la orden k. El segundo término busca minimizar la suma del tiempo de finalización del procesamiento de los trabajos. La inclusión de este término en la función objetivo intenta recoger la frecuente inquietud de las empresas por disminuir los tiempos ociosos de las máquinas.

3.8. Resultados

Los resultados del modelo han resultado satisfactorios, pues han mostrado un alto grado de congruencia con respecto a lo esperado. Sin embargo se ha notado que el tiempo computacional pareciera ser exponencial respecto al número de trabajos. Para estos cálculo se utilizaron *solvers* comerciales de programación lineal. Una heurística parece entonces una salida razonable.

En la figura 3.1 se muestra la visualización de un resultado obtenido mediante el modelo descrito anteriormente. En este caso se trata de una instancia *test* con tan sólo 15 trabajos ($N_1, ..., N_{15}$), 4 órdenes ($K_1, ..., K_4$) y 2 líneas (L_1 y L_2). En el eje de las abscisas se representa el tiempo. El largo de procesamiento de los trabajos en la máquinas está representado por el ancho de los rectángulos. Entre dos trabajos sucesivos, se grafica el tiempo de *set up*. Notemos que el último trabajo se termina en el instante 31 y corresponde a la órden -o cliente- K_4. El tiempo computacional para este ejemplo fue de 4 minutos y 34 segundos para una tolerancia de 5 %.

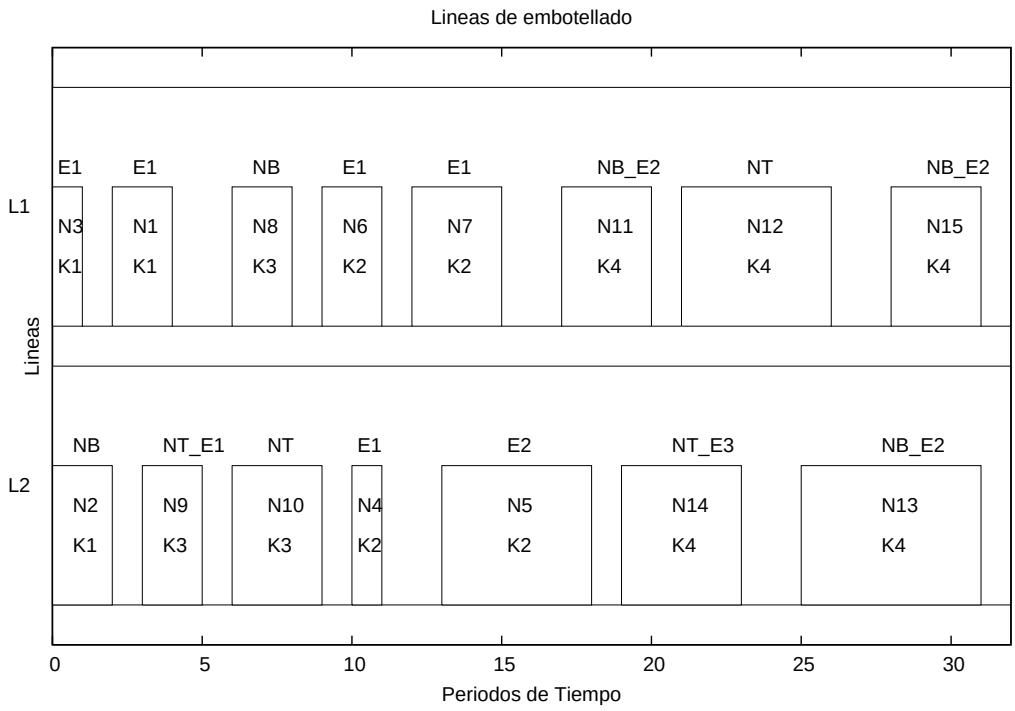

Figura 3.1: Resultado MIP para $n = 15$

Observaciones

- Todas los casos se realizaron en un computador con sistema operativo Windows 7 con procesador Inter(R) Core(TM) i5-2430 CPU @ 2.40Ghz y con 8Gb de memoria RAM.
- Para el MIP se utilizó CPLEX 12.4 mediante el lenguaje GAMS.

3.9. Modelo de minimización de tiempos muertos

En muchas empresas existe la creencia -muchas veces errónea- que el proceso productivo es más eficiente cuando el tiempo de los tiempos muertos o de ocio de las máquinas es mínimo. Para responder a esta inquietud de la industria, es posible modificar el modelo presentado

Numero de Trabajos	Tiempo Resolución MIP
10	1 sec
12	12 sec
15	4 min 34 seg
20	4 hrs
30	>1 dia

Tabla 3.1: Tiempos computacionales MIP

anteriormente en este capítulo de manera de atacar este objetivo. Para lograr lo anterior nos valdremos de una nueva variable:

- $\delta_{n,n'} \in \{0,1\}$: Vale $g_{n'} - o_n$ si $yj_{n',n} = 1$ y 0 en caso contrario.

Notemos que la variable $\delta_{n,n'}$ mide el tiempo que transcurre entre que se termina el trabajo n y se comienza con el siguiente trabajo n'. La nueva función objetivo será entonces la suma de los tiempos de ocio:

$$\text{mín} \sum_{n,n' \in N} \delta_{n,n'}$$

Para que la variable $\delta_{n,n'}$ esté correctamente definida debemos incorporar al modelo dos nuevas restricciones:

$$(26) \quad \delta_{n,n'} \leq M \cdot yj_{n',n} \quad \forall n, n' \in N$$

$$(27) \quad g_{n'} - o_n \leq \delta_{n,n'} + M \cdot (1 - yj_{n',n}) \quad \forall n, n' \in N$$

Los resultados para este modelo siguen el mismo comportamiento que para el modelo anterior. Es decir, los resultados son coherentes con lo esperado pero los tiempos computacionales son altos. De hecho se observa en este caso que los tiempos computacionales son aún más altos. Esto se debe a que se incluyen n^2 nuevas variables y $2n^2$ restricciones nuevas.

Numero de Trabajos	MIP Tiempos Muertos	MIP Standard
10	1 seg	1 seg
12	35 seg	12 seg
15	7 min 21 seg	4 min 34 seg
20	8 hrs	4 hrs
30	>1 día	>1 día

Tabla 3.2: Tiempos computacionales modelo minimización tiempos muertos

Capítulo 4

Heurísticas usando *Constraint Programming*

4.1. Sobre el *Constraint Programming*

El *Constraint Programming* es un poderoso paradigma para resolver problemas de búsqueda combinatoriales, basado en una gran variedad de técnicas distintas, provenientes de diversas áreas como la computación, la programación matemática o la investigación de operaciones. La noción de *Constraint Programming* como término unificador de todas estas técnicas aparece durante la decada de 1980 a pesar que los conceptos de restricción o *constraint* y programación ya eran conocidos desde hace décadas. No se trata, por ende, de un área totalmente nueva, pues la mayoría de los problemas (y a veces soluciones) tratados por el *Constraint Programming* son conocidos hace tiempo. Se trata, más bien, de una lista de herramientas que permiten resolver problemas relativamente específicos, utilizando implementaciones computacionales de algoritmos adhoc.

El *Constraint Programming* utiliza restricciones declaradas a través de variables de decisión definidas en conjuntos predeterminadas. La mayoría de las técnicas usadas en *Constraint Programming* suponen que estos conjuntos son discretos (por ejemplo $\mathbb{Z}, \mathbb{N}, \{0, 1\}$).

El objetivo es definir un verificador de factiblidad que permita testear de manera rápida si la solución es o no factible para el problema en cuestión. En este punto, es de suma importancia la velocidad computacional del verificador, pues en la técnica del *Constraint*

Programming se realizan miles -y a veces millones- de verificaciones, antes de encontrar solución factible.
Otro punto clave para el éxito de esta estrategia es realizar la búsqueda, dentro del conjunto factible, de una manera inteligente. En efecto, en la mayoria de los casos y problemáticas de la vida real, los tamaño a los cuales nos vemos enfrentados pueden fácilmente superar los millones de variables. Por ende si consideramos todas las combinaciones entre variables podemos llegar a casos con miles de millones de combinaciones posibles. Por lo mismo, es clave para el éxito del método que la búsqueda se haga por sectores del conjunto factible más propensos a tener soluciones factibles de buena calidad.

4.1.1. *Constraint Satisfaction*: Un problema clave

El *Constraint Satisfaction* consiste basicamente en encontrar valores para un set de variables decisionales que sean compatibles con un set de restricciones. Como ejemplo simple, podemos mencionar la tarea de escoger componentes para el armado de una bicicleta, como las ruedas, frenos, cadena y marco que sean mutuamente compatible. *Constraint Satisfaction*, como la mayoría de los campos de la inteligencia artificial, puede ser separado en dos miradas que se entrecruzan: representación y razonamiento. A pesar que *Constraint Satisfaction* ha intentado ser encajonado como una tópico dentro de las teorías de búsqueda (razonamiento), su real importancia proviene desde las múltiples aplicaciones reales que tiene (representación).

Los problemas de *Constraint Satisfaction* han sido atacados utilizando múltiples estrategias, desde la teoría de autómatas hasta *ant algorithms* utilizando herramientas matemáticas computacionales de alto nivel.

Backtrack es el método fundamental de búsqueda exacta para *Constraint Satisfaction problems* en el sentido que la solución está garantizada si es que el problema es factible. Uno de los primeros papers en esta área es el de Golomb y Baumert llamado simplemente *Backtrack programming* (1965)[14]. Sin embargo el algoritmo *Backtrack* habia sido ya descubierto de manera independiente por varios autores.

Backtrack construye una solución parcial escogiendo valores para variables hasta que llegue a un camino sin salida en el cual la solución parcial no pueda ser consistentemente extendida. Cuando se alcanza este camino sin salida, se deshacen las últimas asignaciones y se intenta con otras. Esto se hace de manera automática garantizando que finalmente todas

las soluciones sean buscadas. Esta método es mejor que testear todas las soluciones, pues una vez que una solución parcial se desecha, se desechan también todas las posibilidades que incluyan esa solución parcial. Es decir la cantidad de veces que se debe verificar factibilidad es dramaticamente menor que probar todas las posibilidades.

La búsqueda *Backtrack* se representa usualmente mediante un árbol donde cada nodo representa la elección de una variable y cada rama representa un candidato a solución parcial (ver figura 4.1)

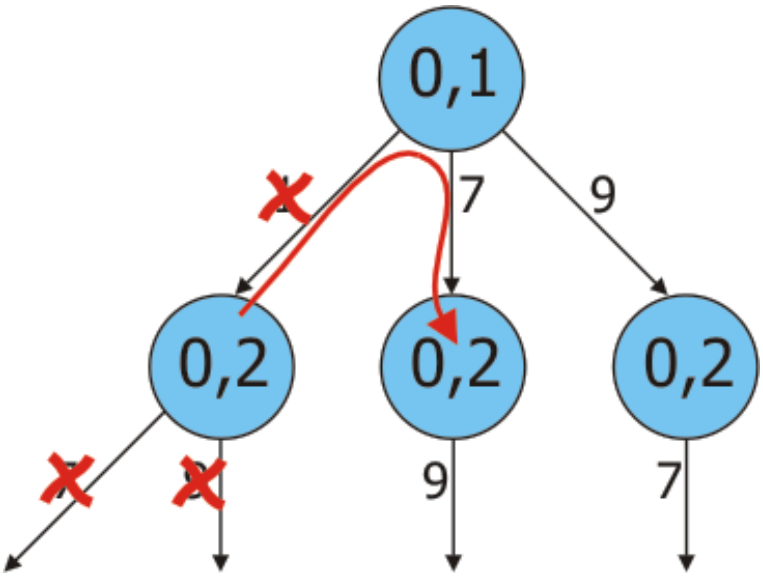

Figura 4.1: Árbol de búsqueda tipo *Backtrack*

Para que la búsqueda sea exitosa -en términos de tiempo- es imprescindible la abilidad e ingenio del programador. Este hecho era ya visualizada en 1965 por Golomb y Baumert:

"Thus the success or failure of backtrack often depends on the skill and ingenuity of the programmer in his ability to adapt the basic methods to the problem at hand and in his ability to reformulate the problem so as to exploit the characteristics of his own computing device. That is, backtrack programming (as many other types of programming) is somewhat of an art."

4.2. Algoritmo Glotón Usando *Constraint Programming*

En esta sección se presenta un método para encontrar una solución factible al problema, considerando la restricción dura que todos los trabajos deben ser programados a tiempo. Esto, en muchos casos, puede satisfacer los requerimientos del cliente considerando que el objetivo sea satisfacer la demanda sin consideraciones adicionales. Para lograr nuestra meta, nos valdremos del *Constraint Programming* y más específicamente del software *Comet*.

Para hacer que el problema se enmarque dentro de la esfera de acción del *Constraint Programming* consideraremos el siguiente conjunto de restricciones, el cual varía con respecto al del capitulo anterior, esencialmente en que las variables se han definido sobre conjuntos discretos y se ha incorporado la restricción dura de satisfacción de la demanda.

4.2.1. Hipótesis de trabajo

- Se consideran varios tipos de productos.
- Los tiempos de setup sólo dependen del trabajo inicial, final y de la línea en que se hace el trabajo.
- Los tiempos para cada trabajo sólo dependen de la línea.
- Los trabajos se pueden realizar con una y una sóla entrega (insumo).

4.2.2. Conjuntos de interés

- K: Conjunto de órdenes o clientes.
- N: Conjunto de trabajos individuales e indivisibles.
- P: Conjunto de productos o tipos de vinos.
- L: Conjunto de líneas.
- E: Conjunto de entregas o recursos disponibles.

4.2.3. Datos de entrada

Constantes

- M: Tiempo máximo de planificación.
- $setup_{n,n',l}$: Tiempo de setup necesario para pasar del trabajo n al trabajo n' en la línea l.

Trabajos

- $tin_{n,p} \in \{0,1\}$: Toma el valor 1 si el trabajo n es del producto p y 0 si no.
- $h_n \in \mathbb{N}$: Tiempo a partir del cual están disponibles los insumos para procesar el trabajo n.
- $rc_{l,n} \in \mathbb{N}$: Tiempo que se demoraría el trabajo n si es procesado en la línea l.
- $lt_n \in \mathbb{N}$: Cantidad de litros del trabajo n.

Órdenes

- $I_k \subseteq N$: Trabajos asociados a la orden k.
- $t_k \in \mathbb{N}$: Tiempo en el cual es requerido que esté lista la orden k.

Entregas

- $ti_e \in \mathbb{N}$: Tiempo inicial de la entrega e.
- $tf_e \in \mathbb{N}$: Tiempo final de la entrega e.
- $vol_e \in \mathbb{N}$: Cantidad de litros de la entrega e.
- $tinto_{e,p} \in \{0,1\}$: Toma el valor 1 ssi la entrega e es del producto p.

4.2.4. Variables

- $g_n \in \mathbb{N}$: Tiempo en el que se empieza a procesar trabajo n.
- $x_{n,l} \in \{0,1\}$: Toma el valor 1 ssi el trabajo n se precesa en la línea l.

- $y_{n,n'} \in \{0,1\}$: Toma el valor 1 ssi el trabajo n se procesa después que el trabajo n' en la misma línea.
- $yj_{n,n'} \in \{0,1\}$: Toma el valor 1 ssi el trabajo n se procesa justo despues del trabajo n' en la misma linea
- $zb_{n,e} \in \{0,1\}$: Toma el valor 1 ssi el trabajo n se realiza con la entrega e.
- $o_n \in \mathbb{N}$: Tiempo en el que se completa el trabajo n.

4.2.5. Restricciones

(1)	$\sum_{l \in L} x_{n,l}$	$=$	1	$\forall n \in N$
(2)	g_n	$\geq$	h_n	$\forall n \in N$
(3)	o_n	$=$	$g_n + \sum_{l \in L} x_{n,l} \cdot rc_{n,l}$	$\forall n \in N$
(4)	$y_{n,n'}$	$\leq$	$1 + \frac{g_n - g_{n'}}{M+1}$	$\forall n, n' \in N, n \neq n'$
(5)	$y_{n,n'}$	$\leq$	$1 - x_{n,l} + x_{n',l}$	$\forall n, n' \in N, n \neq n', \forall l \in L$
(6)	$y_{n,n'}$	$\geq$	$x_{n,l} + x_{n',l} - 2 + \frac{g_n - g_{n'}}{M+1}$	$\forall n, n' \in N, n \neq n', \forall l \in L$
(7)	1	$\geq$	$x_{n,l} + x_{n',l} - y_{n,n'} - y_{n',n}$	$\forall n, n' \in N, n > n'. \forall l \in L$
(8)	$o_{n'}$	$\leq$	$2M(1 - yj_{n,n'}) + 2M(2 - x_{n,l} - x_{n',l}) + g_n - setup_{n',n,l}$	$\forall n, n' \in N, n \neq n', \forall l \in L$
(9)	$\sum_{e \in E} zb_{n,e}$	$=$	1	$\forall n \in N$
(10)	$tin_{n,p} \cdot zb_{n,e}$	$\leq$	$tinto_{e,p}$	$\forall n \in N, \forall e \in E, \forall p \in P$
(11)	$(1 - tin_{n,p}) \cdot zb_{n,e}$	$\leq$	$1 - tinto_{e,p}$	$\forall n \in N, \forall e \in E, \forall p \in P$
(12)	o_n	$\leq$	$tf_e \cdot zb_{n,e} + M(1 - zb_{n,e})$	$\forall n \in N, \forall e \in E$
(13)	$ti_e \cdot zb_{n,e}$	$\leq$	g_n	$\forall n \in N, \forall e \in E$
(14)	$\sum_{n \in N} lt_n \cdot zb_{n,e}$	$\leq$	vol_e	$\forall e \in E$
(15)	o_n	$\leq$	t_k	$\forall n \in I_k$
(16)	$yj_{n',n}$	$\leq$	$y_{n',n}$	$\forall n, n' \in N, n \neq n'$
(17)	$yj_{n,n'}$	$\leq$	$2 - y_{n,n''} - y_{n'',n'}$	$\forall n, n', n'' \in N, n \neq n' \neq n''$
(18)	$y_{n,n'}$	$\leq$	$\sum_{n'' \in N} yj_{n,n''}$	$\forall n, n' \in N, n \neq n'$
(19)	$yj_{n,n}$	$=$	0	$\forall n \in N$

Análisis de las restricciones

A continuación se analizarán las restricciones del modelo.

(1) Un trabajo se asigna a una y solo una línea.

(2) Para empezar a realizar un trabajo deben estar los insumos correspondientes.

(3) Definición de o_n como el tiempo final de realización del trabajo n.

(4) Definición de $y_{n,n'}$. Se obliga que $y_{n,n'}$ valga cero en caso que el inicio del trabajo n' sea posterior a n.

(5) Definición de $y_{n,n'}$. Se obliga que $y_{n,n'}$ valga cero si n' y n no se procesan en la misma línea.

(6) Definición de $y_{n,n'}$. Se obliga que $y_{n,n'}$ valga uno si los trabajos se hacen en la misma línea con n posterior a n'.

(7) Definición de $y_{n,n'}$. Si ambos trabajos están en una misma línea entonces uno y sólo uno de los y vale uno.

(8) Restricción por tiempos de *set up*.

(9) Un trabajo se asigna a una y sola una entrega.

(10) Una entrega se puede solo asociar a un trabajo del mismo producto.

(11) Una entrega se puede solo asociar a un trabajo del mismo producto.

(12) Si un trabajo se asocia a una entrega, el trabajo debe terminar antes del final de la entrega.

(13) Si un trabajo se asocia a una entrega, el trabajo debe empezar después del inicio de la entrega.

(14) La suma de los volúmenes de los trabajos asociados a una entrega no puede sobrepasar su capacidad.

(15) Los trabajos deben estar finalizados antes que expire la fecha de despacho de la orden a la cual pertenecen.

(16) Definición de $yj_{n,n'}$. Si el trabajo n no se procesa después que el trabajo n' entonces tampoco se procesa justo después.

(17) Definición de $yj_{n,n'}$. Solo una de las variables $yj_{n,n'}$ e $yj_{n',n}$ vale uno.

(18) Definición de $yj_{n,n'}$. Para cada par de trabajos n y n' si no hay trabajo n'' que este justo después de n entonces $yj_{n,n'} = 0$.

(19) Definición de $yj_{n,n'}$. Un trabajo no se procesa justo después que sí mismo.

4.2.6. Diferencias con el modelo MIP

En el capítulo 2 se presenta un modelo de programación lineal mixto de similares características al modelo presentado en esta sección. El principal cambio radica en la necesidad que todas las variables decisionales sean definidas sobre conjuntos discretos. Esta condición proviene de los requerimientos básicos de la técnica *Constraint Programming*. Para lograr lo anterior se han incorporado las siguientes modificaciones al modelo MIP:

- Eliminación de la variable decisional $z_{n,e}$ que indicaba el porcentaje sobre el volumen total del trabajo n que aportaba la entrega e. Con lo anterior se asumirá que a cada trabajo se le puede asociar una y sólo una entrega.
- Eliminación de la variable u_k definida como la parte positiva de la diferencia $f_k - t_k$. Lo anterior impone la condición dura que fuerza a realizar todos los trabajos antes del *deadline* o tiempo máximo de despacho de la órden a la cual pertenece.
- Eliminación de las restricciones asociadas $z_{n,e}$ y u_k.
- No se considera función objetivo. En esta sección, el objetivo es encontrar solución factible.

4.2.7. Pseudocódigo

A continuación presentamos el pseudocódigo del Algoritmo Glotón Usando Constraint Programming (AGUCP). Las variables utilizadas en este pseudocódigo son las mismas que las presentadas en el modelos de la sección 5.2.2. El programa *Comet* -en el cual fue programado la heurística- funciona como sigue:

- El usuario define un set de restricciones. En este caso, las restricciones usadas están definidas en la sección 4.2.5.
- Se define una secuencia en la cual se visitarán los distintos puntos del conjunto factible. Tal como se mencionó en la sección 4.1, es de suma importancia que la búsqueda sea realizada de una manera inteligente, utilizando las particularidades del problema en cuestión. En este caso, la idea general es asignar aleatoriamente cada trabajo a cada línea. Una vez realizado lo anterior, el algoritmo intenta colocar el trabajo lo antes posible siempre teniendo cuidado de satisfacer las restricciones 5.2.2. La estocasticidad del modelo tiene como consecuencia que el programa pueda entregar soluciones distintas en distinas corridas, aún cuando los datos sean lo mismo. Lo anterior es útil para el caso en el que no se logre encontrar solución, pues existe la posibilidad que ejecutando el programa nuevamente si se logre tener solución.
- Cuando *Comet* intenta asignar a una variable un cierto valor, y este valor es infactible, inmediatamente intenta con otro valor siguiendo el esquema definido por el algoritmo.
- Si al final *Comet* constata que, para una cierta variable, no es posible asignarle ningún valor, entonces no entregará solución. Este hecho, no es del todo deseable, por cuanto, a pesar que hayan uno más trabajos que no puedan ser agendados, parece sensato obtener a lo menos una solución parcial. Este problema será atacado y resuelto con el algoritmo AGUCP++ definido en la sección 4.3.6.

Algorithm 1 Algoritmo Glotón Usando *Constraint Programming* (**AGUCP**)

Ordenar N según urgencia de los trabajos
$W \leftarrow N$
while $W \neq \emptyset$ **do**
  Seleccionar primer elemento $w \in W$
  Seleccionar aleatoriamente $l \in L$
  try: $x_{w,l} = 1$
  elsetry: $x_{w,l} = 0$
  $W \leftarrow W \setminus \{w\}$
end while
$W \leftarrow N$
while $W \neq \emptyset$ **do**
  Seleccionar primer elemento $w \in W$
  $m = 1$
  while TRUE **do**
    try: $g_w = m$, **break**
    elsetry: $g_w > m$, $m \leftarrow m + 1$
  end while
  $W \leftarrow W \setminus \{w\}$
end while
Seleccionar aleatoriamente una entrega factible para cada trabajo

4.2.8. Observaciones

- El comando *try* intenta incluir la restricción adicional que le sigue. Si hay infactibilidad incluye la restricción adicional que le sigue al comando *elsetry*.
- Las variables $yj_{n,n}$ son prescindibles en el modelo, sin embargo su inclusión se mantiene debido a que los tiempos computacionales son menores, en ciertas instancias, cuando están presentes.
- El algoritmo es estocástico pues en la línea 5 y 20 se hace una elección al azar.
- La línea 20 se hace utilizando el comando *label* en el programa *Comet*.

4.2.9. Resultados

En la mayoría de los casos expuestos el algoritmo ha encontrado solución de manera inmediata. Sin embargo en alrededor del 15 % de los casos el algoritmo no logra encontrar solución.
A continuación se presenta una tabla de comparación entre el modelo de optimización lineal mixto y el algoritmo glotón usando *Constraint programming* (los tiempos se muestran en segundos).

Observaciones

- Todas los casos se realizaron en un computador con sistema operativo Windows 7 con procesador Inter(R) Core(TM) i5-2430 CPU @ 2.40Ghz y con 8Gb de memoria RAM.
- Para el MIP se utilizó CPLEX 12.4 mediante el lenguaje GAMS.
- Para el MIP se consideró como tiempo final, el primer momento en el cual se encuentra solución factible.
- Para AGUCP se utilizó el programa *Comet*.
- Del total de casos estudiados hsata el momento en un 15 % AGUCP no encuentra solución.
- El código del programa en *Comet* se puede encontrar en el APÉNDICE parte B

Numero de Trabajos	Tiempo Resolución MIP	Tiempo Resolución AGUCP
10	1 sec	1 sec
12	12 sec	2 sec
15	4 min 34 sec	1 sec
20	4 hrs	1 sec
30	>1 día	1 sec

Tabla 4.1: Tiempos computacionales AGUCP vs MIP

Instancias de mayor tamaño

A continuación utilizaremos AGUCP para instancias de tamaño superior, es decir, instancias no resolubles utilizando la estrategía MIP. Para poder *testear* el algoritmo para distintos valores de n utilizaremos un programa computacional en *python* que genere datos aleatorios a partir de ciertos *imputs* que se le entreguen (numero de trabajos, trabajos por orden etc...). El código del programa se encuentra en APÉNDICE parte C. Los resultados son los siguientes:

Numero de Trabajos	Tiempo Resolución AGUCP
30	0.7 sec
35	1.5 sec
40	1.4 sec
45	2.4 sec
50	3.1 sec
60	5.4 sec
70	7.8 sec
80	12.3 sec
90	16.7 sec
100	24.0 sec
110	34.6 sec
120	44.6 sec

Inspirados en la tendencia de los puntos, parece sensato utilizar una regresión cuadrática. Buscamos encontrar coeficientes a, b, c tal que:

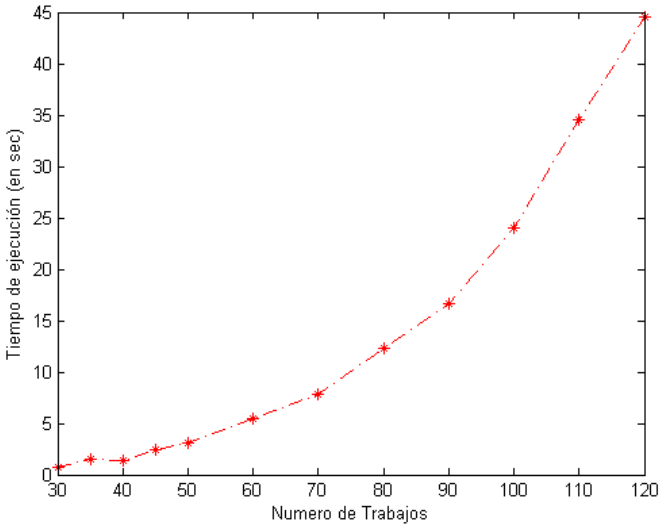

Figura 4.2: Rendimiento computacional de AGUCP

$$t(n) \approx an^2 + bn + c$$

Usando el comando *polyfit* de *MATLAB* obtenemos:

```
>> polyfit(n,t,2)

ans =

    0.0064   -0.5020   11.2203
```

Lo que expresado gráficamente queda como sigue:

La figura 4.3 muestra que el tiempo computacional en función del número de trabajos es interpolable por un polinomio cuadrado. Por lo tanto, podemos inferir que la complejidad computacional de AGUCP es $o(n^2)$, donde n es el número de trabajos. Esto permite augurar buenos resultados en términos de tiempo de computación para instancias de mayor tamaño.

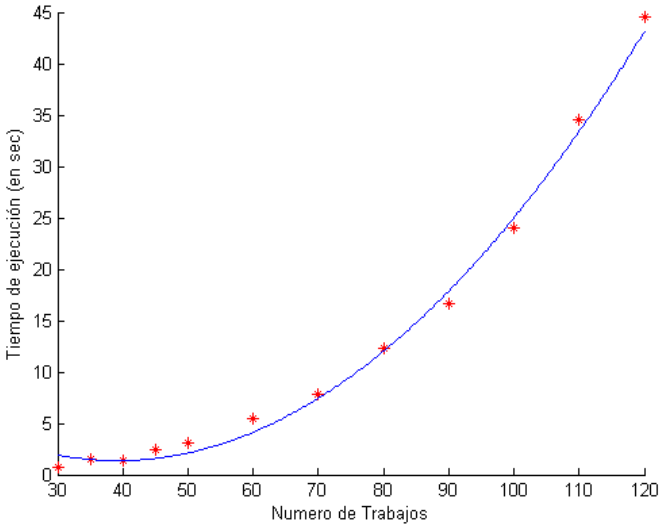

Figura 4.3: Regresión cuadrática

4.3. Algoritmo Glotón Usando *Constraint Programming*++

En la sección anterior, se presentó el *Algoritmo Glotón Usando Constraint Programming* (AGUCP) que busca encontrar solución factible de manera eficiente al problema de "Programación de Trabajos en Líneas de Producción"(P). Los resultados de dicho algoritmo, programado en *Comet*, fueron buenos pues permitieron resolver instancias reales en tiempos razonables, lo que no había sido posible hasta entonces usando la estrategia MIP. Sin embargo, el algoritmo no encontraba solución en alrededor de un 10 % de los casos lo que complicaba su uso de manera industrial. Por otra lado la utilización de *Comet* no permitía avisorar mejoras simples al algoritmo pues se trata de un software empaquetado y sin acceso a los códigos fuente. En esta sección, por la tanto, se describe una implementación propia en Python del algoritmo que llamaremos AGUCP++ el cual incluye mejoras sustanciales cuyos resultados se expondrán en la presente sección.

4.3.1. Un modelo adaptado al *Constraint Programming*

Una de las debilidades de la programación anterior de AGUCP era el uso de variables y restricciones poco adecuadas al espirítu del *Constraint Programming* (e.g Variables Binarias). Si bien el nuevo modelo conserva varias similitudes con el modelo para AGUCP, esta vez se

intentó identificar y utilizar directamente las variables de decisión, a saber:

- Línea en la cual se procesa cada trabajo.
- Tiempo en el cual se inicia el proceso de cada trabajo.
- Entrega con la que se procesa cada trabajo

4.3.2. Hipótesis de Trabajo

- Se consideran varios tipos de productos.
- Los tiempos de setup sólo dependen del trabajo inicial, final y de la línea en que se hace el trabajo.
- Los tiempos para cada trabajo solo dependen de la línea.

4.3.3. Conjuntos de Interés

- K: Conjunto de órdenes.
- N: Conjunto de trabajos.
- P: Conjunto de productos.
- L: Conjunto de líneas.
- E: Conjunto de entregas.

4.3.4. Datos de entrada

Constantes

- M: Horizonte de tiempo.
- $setup_{n,n',l}$: Tiempo de setup necesario para pasar del trabajo n al trabajo n' en la línea l.

Trabajos

- $tin_{n,p} \in \{0,1\}$: Toma el valor 1 si el trabajo n es del producto p y 0 si no.
- $h_n \in \mathbb{N}$: Tiempo a partir del cual están disponibles los insumos para procesar el trabajo n.
- $rc_{l,n} \in \mathbb{N}$: Tiempo que se demoraría el trabajo n si es procesado en la línea l.
- $lt_n \in \mathbb{N}$: Cantidad de litros del trabajo n.

Órdenes

- $I_k \subseteq N$: Trabajos asociados a la orden k.
- $t_k \in \mathbb{N}$: Tiempo en el cual es requerido que esté lista la orden k.

Entregas

- $ti_e \in \mathbb{N}$: Tiempo inicial de la entrega e.
- $tf_e \in \mathbb{N}$: Tiempo final de la entrega e.
- $vol_e \in \mathbb{N}$: Cantidad de litros de la entrega e.
- $tinto_{e,p} \in \{0,1\}$: Toma el valor 1 ssi la entrega e es del producto p.
- $ef_p \subseteq E$: Conjunto de entregas que pertencen al producto p.

4.3.5. Variables de decisión

- $l_n \in L$: Línea con la cual se procesa el trabajo n.
- $g_n \in \{1,2,...,M\}$: Tiempo en el que se empieza a procesar el trabajo n.
- $e_n \in E$: Entrega con la cual se procesa el trabajo n.

Definamos el siguiente conjunto:

$$F = L^{|N|} \times \{1,2,...,M\}^{|N|} \times E^{|N|}$$

4.3.6. Verificación de factibilidad

El algoritmo siguiente busca determinar si una solución $(l, g, e) \in F$ es factible o no para el problema (P). Agregaremos la posibilidad de verificar factibilidades parciales utilizando la siguiente variable:

- $b_n \in \{0, 1, 2, 3\}$: Donde b_n vale 0 si el trabajo n no debe ser considerado. Vale 1 si solo deben considerarse restricciones de entregas. Vale 2 si sólo deben considerarse restricciones de setup e insumos. Y vale 3 si el trabajo n debe ser considerado en su totalidad. Esta funcionalidad extra de la variable b_n no es utilizada en el algorimto AGUCP sin embargo puede ser de utilidad en trabajos futuros.

4.3.7. Objetivo del Modelo

El objetivo del modelo es:

$$\text{Encontrar } (l, g, e) \in F \text{ tal que Verificador}(b^*, l, g, e) = \text{True}$$

donde $b^* = (3, ..., 3)$ o en su defecto b^* con la mayor cantidad de elementos igual a 3.

Nota:

Buscar todas las opciones es computacionalmente imposible. En efecto:

$$|F| = (M \cdot |L| \cdot |E|)^{|N|}$$

Si consideramos una instancia pequeña con $M = 100$, $|L| = 2$, $|E| = 5$, $|N| = 10$ se tiene que $|F| = 10^{13}$!!

4.3.8. AGUCP++

A continuación se presenta el algoritmo AGUCP++, cuyas principales novedades con respecto a la versión anterior son:

- Se trabaja con una menor cantidad de variables debido al modo de guardar la información.

Algorithm 2 Verificador

Input: (b, l, g, e)
for $n \in N$ **do**
  $o_n \leftarrow g_n + rc_{n,l_n}$
  if $(b_n = 2$ **or** $b_n = 3)$ **and** ($tinto_{e_n} \neq tin_n$ **or** $g_n < ti_{e_n}$ **or** $tf_{e_n} < o_n$) **then**
    return False
  end if
  if $(b_n = 1$ **or** $b_n = 3)$ **and** $g_n < h_n$ **then**
    return False
  end if
end for
for $ee \in E$ **do**
  if $\sum_{n \in N, b_n \geq 2, e_n = ee} lt_n < vol_{ee}$ **then**
    return False
  end if
end for
for $k \in K, n \in I_k$ **do**
  if $(b_n = 1$ **or** $b_n = 3)$ **and** $o_n > t_k$ **then**
    return False
  end if
end for
for $c \in L$ **do**
  Ordenar según g_n y definir $orden_c$ lista de trabajos procesados en linea c
  for $i = 1 : \text{length}(orden_c) - 1$ **do**
    $n \leftarrow orden_c[i]$
    $nn \leftarrow orden_c[i+1]$
    if $(b_n = 1$ **or** $b_n = 3)$ **and** $o_n > g_{nn} - setup_{nn,n,c}$ **then**
      return False
    end if
  end for
end for
return True

- Para cada trabajo se escoge la línea en la cual el tiempo de ejecución es el más bajo y no al azar como en AGUCP.
- En caso que el trabajo no sea factible con esa línea, se intenta con el resto.
- El algoritmo entrega una solución, a pesar que, haya uno o más trabajos que no pudieron incorporarse.

Algorithm 3 Algoritmo Glotón Usando *Constraint Programming* ++ (**AGUCP++**)

$b, l, g, e \leftarrow 0$

for $n \in N$ **do**
  Escoger $l_n \in argmin_{c \in L}\ rc_{n,c}$
  $W_n \leftarrow L \setminus l_n$
  $m \leftarrow 1$
  $b_n \leftarrow 3$

  while True **do**
    if $m > M$ **and** $|W_n| > 0$ **then**
      $m \leftarrow 0$
      Escoger $l_n \in W_n$
      $W_n \leftarrow L \setminus l_n$
    end if

    if $m > M$ **and** $|W_n| = 0$ **then**
      $b_n \leftarrow 0$
      Break While
    end if

    for $ee \in ef_{tin_n}$ **do**
      $e_n \leftarrow ee$
      if verificador(b, l, g, e) **then**
        Break While
      end if
    end for

    $m \leftarrow m + 1$

  end while

end for

4.3.9. Resultados

- Todas los casos se realizaron en un computador con sistema operativo Windows 7 con procesador Inter(R) Core(TM) i5-2430 CPU @ 2.40Ghz y con 8Gb de memoria RAM.
- Para el MIP se utilizó CPLEX 12.4 mediante el lenguaje GAMS.
- Para el MIP se consideró como tiempo final, el primer momento en el cual se encuentra solución factible.
- Para AGUCP se utilizó el programa *Comet*.
- Para AGUCP++ se utilizó el lenguaje de programación *Python*.
- Del total de casos estudiados hasta el momento en un 10 % AGUCP no encuentra solución.
- Del total de casos estudiados hasta el momento en un 2 % AGUCP++ no incorpora todos los trabajos, pero de todos modos entrega solución.

Numero de Trabajos	AGUCP++	AGUCP	MIP
30	0.1 seg	0.7 seg	>1 día
35	0.2 seg	1.5 seg	>1 día
40	0.5 seg	1.4 seg	>1 día
45	0.7 seg	2.4 seg	>1 día
50	0.9 seg	3.1 seg	>1 día
60	1.6 seg	5.4 seg	>1 día
70	2.9 seg	7.8 seg	>1 día
80	3.2 seg	12.3 seg	>1 día
90	4.9 seg	16.7 seg	>1 día
100	7.1 seg	24.0 seg	>1 día
110	11.7 seg	34.6 seg	>1 día
120	14.4 seg	44.6 seg	>1 día
200	80.8 seg	256.4 seg	>1 día
300	312.8 seg	1005.6 seg	>1 día
400	1055.7 seg	4325.8 seg	>1 día

4.3.10. Observaciones Finales

Las mejoras del algoritmo y la implementación propia en un lenguaje más amplio como *python*, han tenido como tres grandes consecuencias:

- Disminución de los tiempos computacionales en un 70 % aproximadamente.
- Posibilidad de entregar solución a pesar que no todos los trabajos se hayan agendado.
- Disminución de los casos en los cuales uno o más trabajos no se agendan a un 2 % de los casos estudiados.

Capítulo 5

Mejoramiento de la solución

En este capítulo se buscará mejorar las soluciones encontradas en el capítulo 4, según criterios predefinidos -o en terminos matemáticos, distintas funciones objetivos-. En este punto, es necesario mencionar que la manera en que los algoritmos AGUCP y AGUCP++ están construidos, garantizan encontrar soluciones de una buena calidad por lo que las mejoras no necesariamente serán sustanciales.

Debido a experiencias con empresas que utilizan producción en líneas, nos hemos percatado que un objetivo secundario, después de la satisfacción de la demanda, suelen ser la disminución de los tiempos de ocio de las máquinas y el tiempo final del último trabajo agendado. Definiremos, por ende, los siguientes criterios de optimización:

1) Objetivo Primario: Minimizar trabajos incumplidos.

2) Objetivo Secundario: Minimizar la suma de los tiempos de finalización de los trabajos.

Se optimizará, comparando soluciones, del siguiente modo. En una primera instancia se observará el criterio 1) y nos quedaremos con la solución que tenga mejor *performance*. En caso de empate, se procederá a analizar el criterio 2) y se escogerá el que tenga menor valor de la suma de los tiempos de finalización de los trabajos.

Podemos resumir lo anterior en una sola función objetivo haciendo que el término correspondiente al objetivo primario sea -en valor absoluto- mucho más importante que el término correspondiente al objetivo secundario:

$$M \cdot |N \setminus A| + \sum_{n \in A} o_n$$

donde $A \subseteq N$ es el conjunto de los trabajos agendados

5.1. Busquedas aleatorizadas usando AGUCP++

5.1.1. Algoritmo

Recordemos que el algoritmo AGUCP++ (4.3.6) recorre, secuencialmente, los trabajos de la instancia para ir agendandolos lo antes posible. El orden utilizado hasta ahora, utiliza el t_k-criterio. Es decir primero se agendan los trabajos pertenecientes a ordenes cuyos *deadline* son menores. Si bien el t_k-criterio parece razonable, es posible que existan otras secuencias que permitan mejorar la función objetivo. Para ello incluiremos la posibilidad de realizar varias corridas de AGUCP++ usando secuencias obtenidas a través de permutaciones aletorias de la n-tupla de los trabajos tal que como lo muestra la figura 5.1.

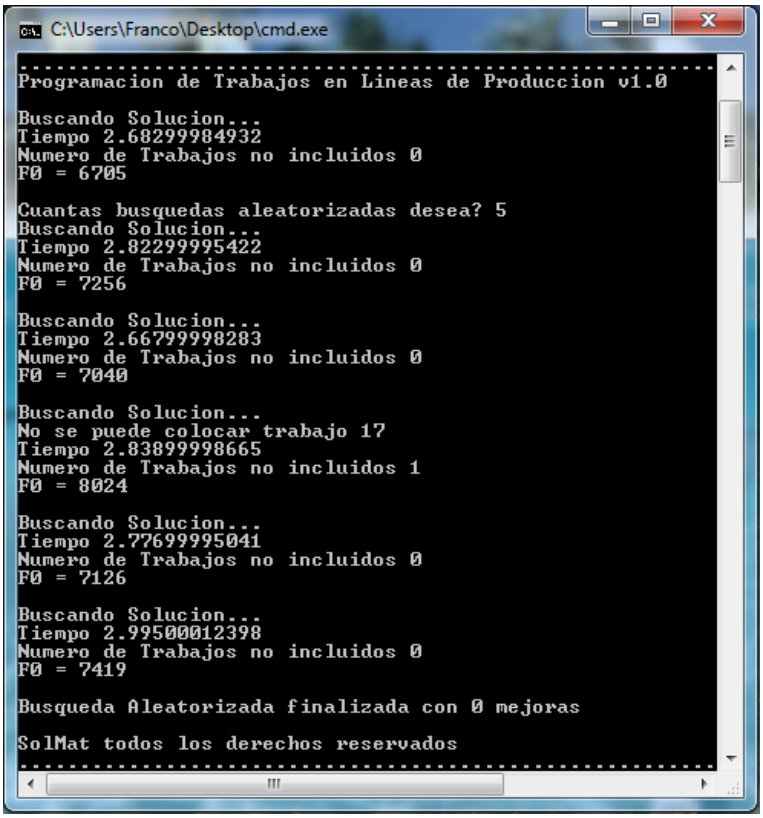

Figura 5.1: Ejemplo de busquedas aletorizadas usando AGUCP++

A continuación se muestra el pseudoalgoritmo:

Algorithm 4 Busquedas aleatorizadas usando AGUCP++

```
Obtener solución s̄ a partir de AGUCP++ normal
Recibir la cantidad m de busquedas aleatorizadas deseadas
for i = 1...m do
    Ordenar N aleatoriamiente
    Obtener solución s a partir AGUCP++ usando dicha secuencia
    if val(s)>val(s̄) then
        s̄ ← s
    end if
end for
Return s̄
```

5.1.2. Resultados

Los instancias probadas hasta ahora utilizando el Algoritmo 4 no han logrado mejorar las soluciones obtenidas utilizando el secuenciamiento dado por el t_k-criterio.

5.2. *Local Search* vertical

5.2.1. Sobre el *Local Search*

En la presente sección se busca obtener mejores soluciones utilizando *Local Search*. Recordemos que esta técnica se basa en el concepto de vecindades. Más especificamente para cada punto x factible, se debe definir un conjunto $V(x)$ de los vecinos de x y utilizar el siguiente procedimiento:

(1) Obtener solución inicial x_0 factible (a través de AGUCP por ejemplo)

(2) En casa iteración calcular x_{n+1} maximizando en $V(x_n)$ (subproblema).

Para que este tipo de heurísticas tenga éxito es de crucial importancia definir $V(x)$ de modo que:

- Existan mejoras de cierta relevancia.
- Cada subproblema de optimización

$$(P_n) \quad \max_{x \in V(x_n)} fo$$

se resuelva rápidamente.

5.2.2. Modelo de optimización

El modelo de optimización que se utilizará en todos las búsquedas locales corresponde es similar al expuesto en la sección AGUCP incluyendo la función objetivo deseada:

Hipótesis de trabajo

- Se consideran varios tipos de productos.
- Los tiempos de setup sólo dependen del trabajo inicial, final y de la línea en que se hace el trabajo.
- Los tiempos para cada trabajo sólo dependen de la línea.
- Cada trabajo puede realizarse con una y sólo una entrega.

Conjuntos de interés

- K: Conjunto de órdenes o clientes.
- N: Conjunto de trabajos individuales e indivisibles.
- P: Conjunto de productos o tipos de vinos.
- L: Conjunto de líneas.
- E: Conjunto de entregas o recursos disponibles.

Datos de entrada

Constantes

- M: Tiempo máximo de planificación.
- $setup_{n,n',l}$: Tiempo de setup necesario para pasar del trabajo n al trabajo n' en la línea l.

Trabajos

- $tin_{n,p} \in \{0,1\}$: Toma el valor 1 si el trabajo n es del producto p y 0 si no.
- $h_n \in \mathbb{N}$: Tiempo a partir del cual están disponibles los insumos para procesar el trabajo n.
- $rc_{l,n} \in \mathbb{N}$: Tiempo que se demoraría el trabajo n si es procesado en la línea l.
- $lt_n \in \mathbb{N}$: Cantidad de litros del trabajo n.

Órdenes

- $I_k \subseteq N$: Trabajos asociados a la orden k.
- $t_k \in \mathbb{N}$: Tiempo en el cual es requerido que esté lista la orden k.

Entregas

- $ti_e \in \mathbb{N}$: Tiempo inicial de la entrega e.
- $tf_e \in \mathbb{N}$: Tiempo final de la entrega e.
- $vol_e \in \mathbb{N}$: Cantidad de litros de la entrega e.
- $tinto_{e,p} \in \{0,1\}$: Toma el valor 1 ssi la entrega e es del producto p.

Variables

- $g_n \in \mathbb{N}$: Tiempo en el que se empieza a procesar trabajo n.
- $x_{n,l} \in \{0,1\}$: Toma el valor 1 ssi el trabajo n se precesa en la línea l.
- $y_{n,n'} \in \{0,1\}$: Toma el valor 1 ssi el trabajo n se procesa después que el trabajo n' en la misma línea.
- $yj_{n,n'} \in \{0,1\}$: Toma el valor 1 ssi el trabajo n se procesa justo despues del trabajo n' en la misma linea
- $zb_{n,e} \in \{0,1\}$: Toma el valor 1 ssi el trabajo n se realiza con la entrega e.
- $o_n \in \mathbb{N}$: Tiempo en el que se completa el trabajo n.

Restricciones

(1)	$\sum_{l \in L} x_{n,l}$	$=$	1	$\forall n \in N$
(2)	g_n	$\geq$	h_n	$\forall n \in N$
(3)	o_n	$=$	$g_n + \sum_{l \in L} x_{n,l} \cdot rc_{n,l}$	$\forall n \in N$
(4)	$y_{n,n'}$	$\leq$	$1 + \frac{g_n - g_{n'}}{M+1}$	$\forall n, n' \in N, n \neq n'$
(5)	$y_{n,n'}$	$\leq$	$1 - x_{n,l} + x_{n',l}$	$\forall n, n' \in N, n \neq n', \forall l \in L$
(6)	$y_{n,n'}$	$\geq$	$x_{n,l} + x_{n',l} - 2 + \frac{g_n - g_{n'}}{M+1}$	$\forall n, n' \in N, n \neq n', \forall l \in L$
(7)	1	$\geq$	$x_{n,l} + x_{n',l} - y_{n,n'} - y_{n',n}$	$\forall n, n' \in N, n > n'. \forall l \in L$
(8)	$o_{n'}$	$\leq$	$2M(1 - yj_{n,n'}) + 2M(2 - x_{n,l} - x_{n',l}) + g_n - setup_{n',n,l}$	$\forall n, n' \in N, n \neq n', \forall l \in L$
(9)	$\sum_{e \in E} zb_{n,e}$	$=$	1	$\forall n \in N$
(10)	$tin_{n,p} \cdot zb_{n,e}$	$\leq$	$tinto_{e,p}$	$\forall n \in N, \forall e \in E, \forall p \in P$
(11)	$(1 - tin_{n,p}) \cdot zb_{n,e}$	$\leq$	$1 - tinto_{e,p}$	$\forall n \in N, \forall e \in E, \forall p \in P$
(12)	o_n	$\leq$	$tf_e \cdot zb_{n,e} + M(1 - zb_{n,e})$	$\forall n \in N, \forall e \in E$
(13)	$ti_e \cdot zb_{n,e}$	$\leq$	g_n	$\forall n \in N, \forall e \in E$
(14)	$\sum_{n \in N} lt_n \cdot zb_{n,e}$	$\leq$	vol_e	$\forall e \in E$
(15)	o_n	$\leq$	t_k	$\forall n \in I_k$
(16)	$yj_{n',n}$	$\leq$	$y_{n',n}$	$\forall n, n' \in N, n \neq n'$
(17)	$yj_{n,n'}$	$\leq$	$2 - y_{n,n''} - y_{n'',n'}$	$\forall n, n', n'' \in N, n \neq n' \neq n''$
(18)	$y_{n,n'}$	$\leq$	$\sum_{n'' \in N} yj_{n,n''}$	$\forall n, n' \in N, n \neq n'$
(19)	$yj_{n,n}$	$=$	0	$\forall n \in N$

Análisis de las restricciones

A continuación se analizarán las restricciones del modelo.

(1) Un trabajo se asigna a una y sólo una línea.

(2) Para empezar a realizar un trabajo deben estar los insumos correspondientes.

(3) Definición de o_n como el tiempo final de realización del trabajo n.

(4) Definición de $y_{n,n'}$. Se obliga que $y_{n,n'}$ valga cero en caso que el inicio del trabajo n' sea posterior a n.

(5) Definición de $y_{n,n'}$. Se obliga que $y_{n,n'}$ valga cero si n' y n no se procesan en la misma línea.

(6) Definición de $y_{n,n'}$. Se obliga que $y_{n,n'}$ valga uno si los trabajos se hacen en la misma línea con n posterior a n'.

(7) Definición de $y_{n,n'}$. Si ambos trabajos están en una misma línea entonces uno y sólo uno de los y vale uno.

(8) Restricción por tiempos de *set up*.

(9) Un trabajo se asigna a una y sóla una entrega.

(10) Una entrega se puede sólo asociar a un trabajo del mismo producto.

(11) Una entrega se puede sólo asociar a un trabajo del mismo producto.

(12) Si un trabajo se asocia a una entrega, el trabajo debe terminar antes del final de la entrega.

(13) Si un trabajo se asocia a una entrega, el trabajo debe empezar después del inicio de la entrega.

(14) La suma de los volúmenes de los trabajos asociados a una entrega no puede sobrepasar su capacidad.

(15) Los trabajos deben estar finalizados antes que expire la fecha de despacho de la orden a la cual pertenecen.

(16) Definición de $yj_{n,n'}$. Si el trabajo n no se procesa después que el trabajo n' entonces tampoco se procesa justo después.

(17) Definición de $yj_{n,n'}$. Solo una de las variables $yj_{n,n'}$ e $yj_{n',n}$ vale uno.

(18) Definición de $yj_{n,n'}$. Para cada par de trabajos n y n' si no hay trabajo n'' que este justo después de n entonces $yj_{n,n'} = 0$.

(19) Definición de $yj_{n,n'}$. Un trabajo no se procesa justo después que sí mismo.

Función Objetivo

La función objetivo es:

$$\frac{1}{|N|}\sum_{n\in N} o_n$$

Recordemos que la solución inicial obtenida mediante AGUCP++ cumple con la propiedad que todos los trabajos incluidos satisfacen el tiempo máximo permitido o *deadline*. La búsqueda local intenta, por ende, hacer que los trabajos se hagan lo antes posible para que:

(i) Los pedidos sean despachados lo antes posible a los clientes aumentando su satisfacción.

(ii) Crear espacios en las líneas de producción para que los trabajos que, eventualmente, pudieron haber quedado afuera en la iteración 0, tengan la posibilidad de volver a ingresar. Lo anterior puede ocurrir si una ves terminado el proceso de mejoramiento de la solución, volvemos a aplicar AGUCP++ fijando los trabajos que ya ingresaron.

5.2.3. Algoritmo

En nuestro caso, existen varias formas de definir el concepto de vecindad. Nosotros hemos escogido la siguiente definición:

Definición: x e y se dirán vecinos ssi tienen una gran cantidad de trabajos en común en terminos de tiempos de inicio, linea y entrega utilizada.

La función objetivo utilizada en esta oportunidad será:

$$\sum_{n\in A} o_n$$

Algorithm 5 Algoritmo de mejoramiento a través de *Local Search*

1: Obtener solución x_0 usando AGUCP++
2: Definir $nlc \leftarrow int(\frac{|N|}{10})$
3: $H \leftarrow N$
4: **for** $i = 0...(nlc - 1)$ **do**
5: Seleccionar y retirar de H los primeros 10 trabajos para solución x_i
6: Calcular x_{i+1} optimizando y manteniendo fijos los trabajos en H.
7: **end for**
8: Return x_{nlc}

donde $A \subseteq N$ es el conjunto de los trabajos agendados en iteración 0.

5.2.4. Resultados

Los resultados de la heurística basada en el *Local Search* han logrado excelentes resultados logrando mejoras en la función objetivo de entre el 5 % y el 20 %. Según los experimentos realizados hasta el momento, se ha logrado establecer que el porcentaje de mejora depende en gran medida de $|A|$, es decir del número de trabajos considerados en la optimización. Lo anterior es congruente con los datos entregados en la literatura los cuales señalan que el mayor impacto del *Local Search* se logra justamente para medianas y grandes instancias.

A continuación presentamos una tabla resumen de los experimentos realizados hasta el momento. En ella, incluimos el número de trabajos, el tiempo computacional y el porcentaje de mejora de la función objetivo desde entre x_0 y x_{nls}.

Número de Trabajos	Tiempo computacional	Porcentaje de mejora
53	2.3 seg	8 %
100	25.8 seg	12 %
191	237.4 seg	19 %

Tabla 5.1: Tabla resumen algoritmo *Local Search*

5.2.5. Estudio de caso: Instancia 53 Trabajos

Durante esta parte se realizará el análisis de un caso real entregado por una viña chilena para el test de los algoritmos presentados en esta tesis. Las características de los datos

presentados por la viña son:

- Trabajos: 53
- Órdenes: 14
- Líneas: 2
- Productos: 12
- Entregas: 12
- Horizonte de Planificación: 289

En este caso el algoritmo AGUCP++ logra incluir los 53 trabajos en la programación con una función objetivo de 6790. Tras 5 búsquedas locales, la función objetivo se sitúa en 6244 para un gap de 5 %. La evolución de los valores de la función objetivo se muestra en el gráfico 5.2.

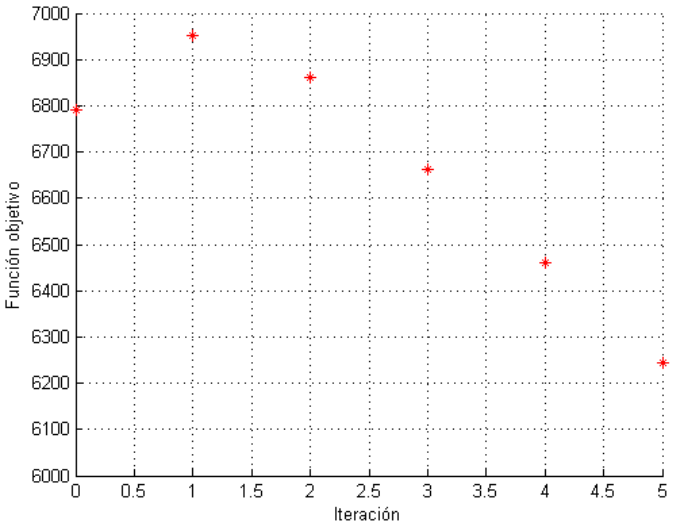

Figura 5.2: Evolución de la función objetivo

Notar que en entre la iteración 0 y 1 de la figura 5.2 se observa un aumento en la función objetivo. Esto, que a primera vista parece un error pues x_0 es solución factible para la

optimización en $V(x_0)$, se explica por el 5 % de tolerancia en el gap para optimización local. En efecto, al utilizar un gap inferior (1 % por ejemplo) se observa que la la funcción objetivo en función del número de iteraciones es decreciente como indica la teoría. Sin embargo la utilización de gap pequeños demora el algoritmo más allá de lo deseable, en especial para instancias de mayor tamaño.

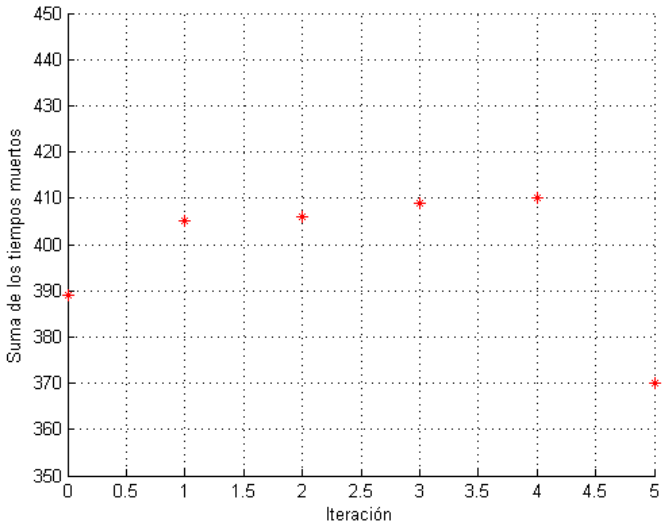

Figura 5.3: Suma de los tiempos muertos

En la figura 5.3 se pueden observar el tiempo total de *set up* propuesto en la solución. Observamos que, al menos en este caso, no existe relación directa, con el función objetivo.

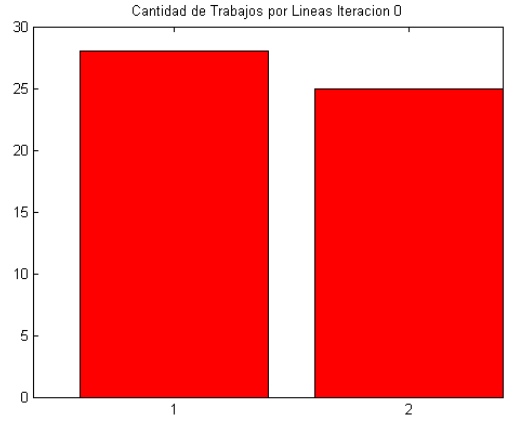

Figura 5.4: Iteración 0: Cantidad de Trabajos por Lineas

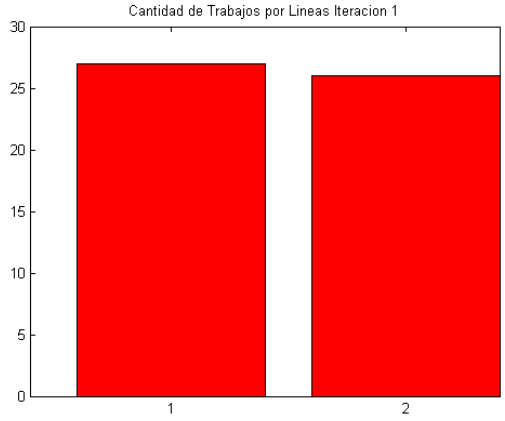

Figura 5.5: Iteración 1: Cantidad de Trabajos por Lineas

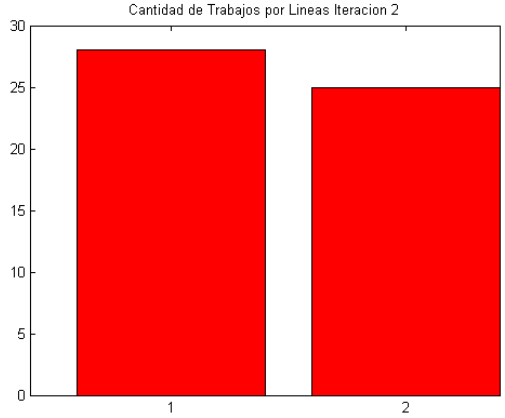

Figura 5.6: Iteración 2: Cantidad de Trabajos por Lineas

5.2.6. Estudio de caso: Instancia 191 Trabajos

Durante esta parte se realizará el análisis de un caso real entregado por una viña chilena para el test de los algoritmos presentados en esta tesis. Las características de los datos presentados por la viña son:

- Trabajos: 191

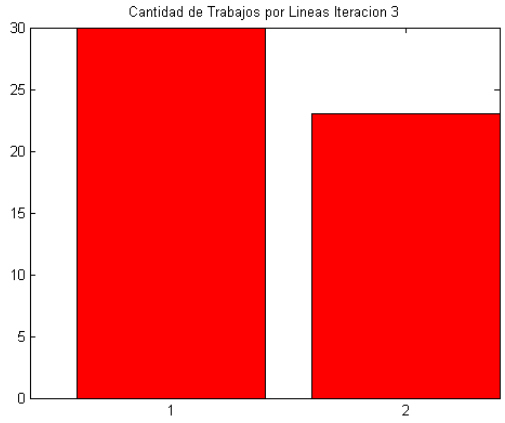

Figura 5.7: Iteración 3: Cantidad de Trabajos por Lineas

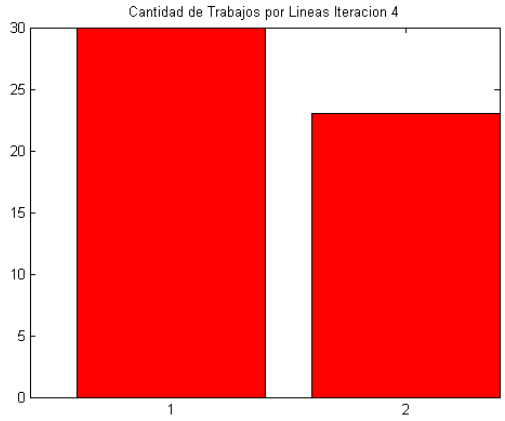

Figura 5.8: Iteración 4: Cantidad de Trabajos por Lineas

- Órdenes: 53
- Líneas: 2
- Productos: 22
- Entregas: 22
- Horizonte de Planificación: 5000

En este caso el algoritmo AGUCP++ logra incluir los 191 trabajos en la programación con una función objetivo de 161879. Tras 5 búsquedas locales, la función objetivo se sitúa en

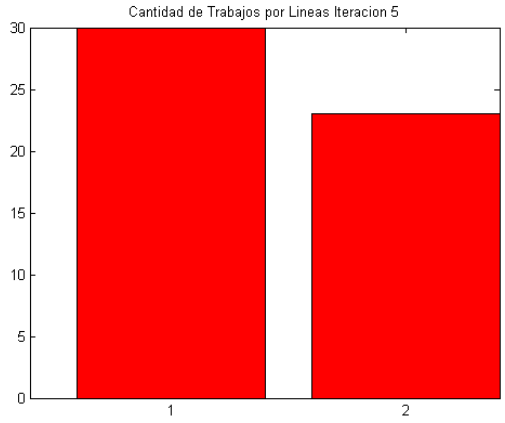

Figura 5.9: Iteración 5: Cantidad de Trabajos por Lineas

130099 para un gap de 5 %. La evolución de los valores de la función objetivo se muestra en el gráfico 5.10.

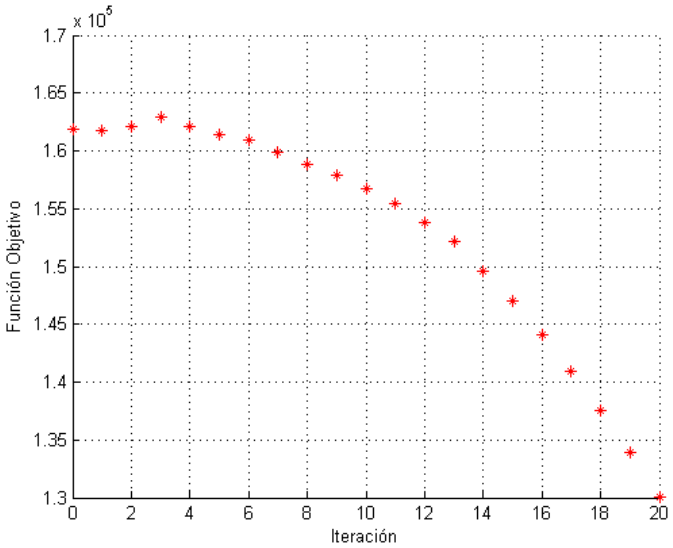

Figura 5.10: Evolución de la función objetivo

Iteración	Cantidad de Trabajos Linea 1	Cantidad de Trabajos Linea 2
0	139	62
1	137	64
2	136	65
3	132	69
4	132	69
5	129	72
6	131	70
7	131	70
8	129	72
9	125	76
10	125	76
11	120	81
12	120	81
13	115	86
14	114	87
15	111	90
16	107	94
17	105	96
18	104	97
19	104	97
20	103	99

Tabla 5.2: Cantidad de trabajos por lineas

En la figura 5.11 se grafica la tabla 5.2 la cual muestra claramente la tendencia a la baja en la cantidad de trabajos procesados en la línea 1 a medida que avanzan las iteraciones del *local search*. Por el contrario, se observa una tendencia a la alza en la línea 2. Este efecto se explica por tres razones esenciales:

(1) AGUCP++ obtiene una solución inicial asignando los trabajos a las líneas más rápidas.

(2) En este ejemplo la línea 1 es casi siempre más rápida que la línea 2.

(3) La función objetivo es la suma de los tiempos de finalización de los trabajos.

Por (1) y (2), la solución inicial (también llamada iteracíón 0) obtenida a través de AGUCP++ privilegia el uso de la línea 1. Esto explica la diferencia de 157 trabajos entre las lineas. A medida que avanzan las iteraciones y en virtud del punto (3) el algoritmo va pasando trabajos paulatinamente desde la línea 1 a la linea 2, de modo de disminuir la función objetivo tal como lo muestra la figura 5.12

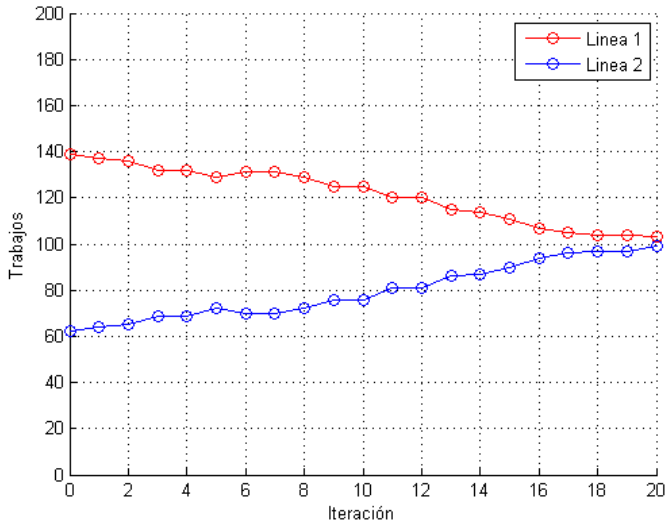

Figura 5.11: Evolución de la cantidad de trabajos por línea

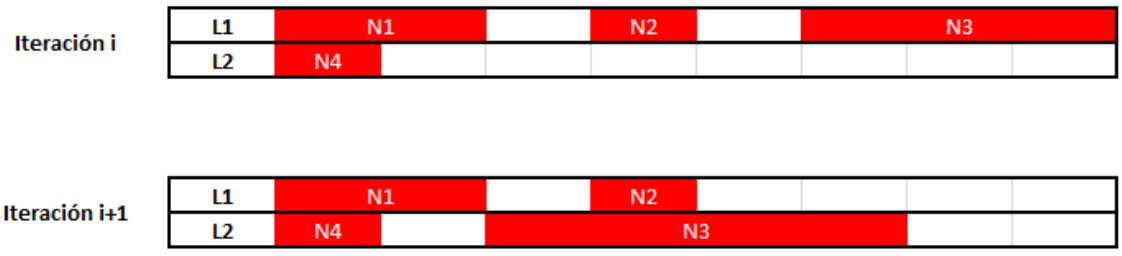

Figura 5.12: Ejemplo de traspaso de trabajos entre lineas

5.3. *Local Search* con solapamiento

En la sección 5.2 se definió una estrategia de búsqueda local en la cual se va optimizando del siguiente modo:

- Se seleccionan los primeros 10 trabajos de la solución inicial obtenida através de AGUCP++ y se fija el resto.
- Se optimiza sobre estos trabajos.
- Se seleccionan los siguientes 10 trabajos, se fija el resto y se optimiza.
- Se repite el proceso anterior hasta haber recorrido todos los trabajos.

Dicho procedimiento permitió obtener mejoras en la función objetivo en un 15 % aproximadamente. Sin embargo tiene la debilidad que no permite hacer alteraciones más drásticas de la solución pues una vez un trabajo optimizado, este no vuelve a participar del proceso de mejora. Además, no tiene la flexibilidad de definir el número de trabajos que se desea utilizar en cada búsqueda local

Para mejorar el punto anterior, se introdujo el concepto de *solapamiento*, a saber, la idea que los trabajos vayan siendo optimizados más de una vez. El algoritmo funciona del siguiente modo:

- Se definen las cantidades η y δ correspondientes respectivamente al número de trabajos a considerar por cada búsqueda local y a la cantidad de trabajos que se solaparán en cada par de iteraciones.
- Se seleccionan los primeros η trabajos, se fija el resto y se optimiza.
- Posteriormente, se avanza en δ trabajos.
- Nuevamente se toman los siguientes η trabajos, se fija el resto y se optimiza.
- Se repite el procedimiento hasta que todos los trabajos hayan sido optimizados.

A continuación se presenta el pseudocódigo del algoritmo de *local search* con solapamiento:

5.3.1. Resultados

El efecto de solapar trabajos a medida que realizan las búsquedas locales permite obtener pequeñas mejoras dentro de la optimización. Lo anterior queda de manifiesto con la siguiente tabla:

Algorithm 6 Algoritmo de mejoramiento a través de *Local Search* con solapamiento

Obtener solución x_0 usando AGUCP++.
Definir η como el número de trabajos a considerar por cada búsqueda local.
Definir δ como la cantidad de trabajos que se solaparán en cada par de iteraciones.
$H \leftarrow N$
while TRUE **do**
  if $H == \emptyset$ **then**
    Break
  end if
  Selecccionar los siguientes η trabajos de H
  Calcular x_i optimizando y manteniendo fijos los trabajos en $N \setminus H$.
  Eliminar los primeros δ trabajos de H entre los η seleccionados.
end while
Return x

Número de Trabajos	Mejora sin solapamiento	Mejora con solapamiento
53	8 %	9 %
100	12 %	14 %
191	19 %	21 %

Tabla 5.3: Efecto del solapamiento

Nota

Para los resultados de la tabla anterior se han utilizado los parámetros $\eta = 10$, $\delta = 5$.

5.4. *Local Search* horizontal

En esta sección describiremos el tercer y último tipo de búsqueda local implementado durante el trabajo de tesis, a saber, el *Local Search* horizonta -o por línea-. Este tipo de búsqueda local está pensado para llevarse a cabo después de realizar el *Local Search* con solapamiento pues su objetivo es optimizar cada de una de las líneas por separado. El algoritmo funciona como sigue:

- Para cada linea hacer lo siguiente.
  - Escoger y optimizar los primeros η trabajos fijando el resto.

- Correrse δ trabajos y repetir paso anterior.
- Hacer este proceso hasta que todos los trabajos hayan sido optimizados.

A continuación se presenta el pseudocódigo del algoritmo *Local Search* por línea:

Algorithm 7 Algoritmo de mejoramiento a través de *Local Search* horizontal

```
1: Obtener solución x_0 usando AGUCP++.
2: Definir η como el número de trabajos a considerar por cada búsqueda local.
3: Definir δ como la cantidad de trabajos que se solaparán en cada par de iteraciones.
4: for l ∈ L do
5:    Definir conjunto H como lista de trabajos en línea l ordenados.
6:    while TRUE do
7:       if H == ∅ then
8:          Break
9:       end if
10:      Selecccionar los siguientes η trabajos de H.
11:      Calcular x_i optimizando sobre los η trabajos seleccionados y dejando el resto fijo
12:      Eliminar los primeros δ trabajos de H.
13:   end while
14: end for
15: Return x
```

5.4.1. Resultados

La realización de este tipo de búsquedas locales por línea ha resultado ser un excelente complemento a las búsquedas locales verticaleso (ptimizando en ambas líneas) presentadas en esta sección. Los tiempos computacionales no aumentan dramaticamente y el porcentaje de mejora sube de manera considerable. A continuación presentamos los resultados para las instancias testeadas:

Número de Trabajos	local search vertical	local search vertical+horizontal
53	9 %	14 %
100	14 %	21 %
191	21 %	28 %

Tabla 5.4: Efecto del solapamiento

Capítulo 6

Conclusiones

La presente tesis aborda el problema de asignar trabajos a líneas de producción de manera de satisfacer un set de restricciones operacionales minimizando el tiempo de despachos de los encargos.

Las estrategias para atacar el problema son dos:

(1) Optimización exacta utilizando un modelo de programación lineal mixta.

(2) Optimización aproximada utilizando un algoritmo para obtener solución factible inicial de calidad aceptable, más un proceso de mejoramiento de la solución a través de una metodología tipo *local search*.

Para la estrategia (1) se logra obtener el óptimo al problema. Sin embargo, lo anterior es sólo posible para instancias de pequeños tamaños, a saber, instancias de máximo 15 trabajos. El cálculo de la sección 4.3.7 demuestra que el problema es exponencial con respecto al número de trabajos. Este hecho explicaría la rápida velocidad con que crece el tiempo computacional a medida que aumenta el número de trabajos. Todo lo anterior nos lleva a tener que utilizar heurísticas para instancias de mayor tamaño.

La estrategia (2) utiliza el denominado Algoritmo Glotón Usando Constraint Programming (AGUCP) y su versión optimizada (AGUCP++) para obtener una primera solución factible al problema. Estos algoritmos funcionan a alta velocidad, logrando resultados en pocos segundos para instancias de hasta 100 trabajos. Para instancias de tamaño superior ($|N|$>300) AGUCP++ logra resultados en tiempos cercanos a los 1000 segundos. Análisis

empíricos demuestran que tanto AGUCP como AGUCP++ son polinomiales en el número de trabajos lo que explica su buen rendimiento computacional.
Finalmente, las soluciones obtenidas utilizando la heurística golosa, son mejoradas utilizando una estrategia de búsqueda local. Este proceso permite obtener soluciones cuya función objetivo mejora entre un 5 % y 28 % obteniendo de este modo soluciones de buena calidad en tiempos razonables

Con respecto a futuros trabajos, una posibilidad que parece interesante es incluir variabilidad a la data o asumir que los pedidos llegan de manera aleatoria en el horizonte de planificación siguiendo algún proceso estocástico (por ejemplo un proceso de Poisson).

Bibliografía

[1] R. Aggoune, "Minimizing the makespan for the flow shop scheduling problem with availability constraints" *, European Journal of Operational Research 153 (2004) 534–543.*

[2] T. Aldowaisan and A. Allahverdi, "New heuristics for m-machine no-wait flowshop to minimize total completion time" *, Omega,32, 345–352, 2004*

[3] H. G. Campbell, R. A. Dudek, M. L. Smith, "A heuristic algorithm for the n job, m machine sequencing problem" *, Management Science, Vol. 16, No. 10, June, 1970.*

[4] G. Cavory, R. Dupas and G. Goncalves, "A genetic approach to the scheduling of preventive maintenance tasks on a single product manufacturing production line" *, Int. J. Production Economics 74 (2001) 135-146.*

[5] R. W. Conway, W. L. Maxwell and L. W. Miller, "Theory of Scheduling" *, Addison- Wesley: Reading, MA, 1967.*

[6] M. Held and R. Karp, "A dynamic programming approach to sequencing problems" *, J. SOC. INDJST. APPL. MATH. Vol. 10, No. 1, March, 1962.*

[7] S. M Johnson, "Optimal two-and three-stage production schedules with set-up times included" *, Naval Res. Logist. Quart, 1, 61–68, 1954*

[8] F. Hillier, R. Boling, "On the optimal allocation of work in symmetrically unbalanced production line systems with variable operation times" *, Management Science, Vol. 25, No. 8, August 1979.*

[9] A. M. Geoffrion and G. W. Graves, "Scheduling Parallel Production Lines with Changeover Costs: Practical Application of a Quadratic Assignment/LP Approach" *, Operations Research, Vol. 24, No. 4, July-August 1976.*

[10] S. Kim and B. Jeong, "Product sequencing problem in Mixed-Model Assembly Line to minimize unfinished works" , *Computers and Industrial Engineering 53 (2007) 206–214.*

[11] Z. Lomnicki, "A branch-and-bound algorithm for the exact solution of the three-machine scheduling problem" , *Operat. Res. Quart., 16, 89–100, 1965.*

[12] T. Prabhakar, "A production scheduling problem with sequencing considerations" , *Management Science Vol. 21, No. 1, September, 1974.*

[13] F. Rossi, P. van Beek and T. Walsh, *Handbook of Constraint Programming*

[14] S. Golomb, L. Baumert "Backtrack programming" , *J. ACM, 12:516–524, 1965.*

[15] Dynamic Decision Technologies Inc, Comet Tutorial *March 2010*

APÉNDICE

Apéndice A

Código en *Python* de creatingdata.py

```
import random

salida=open('data.co','w')

ntrab=50
trabxord=5
nord=ntrab/trabxord
nprod=3
nlin=2
nent=10

hor=ntrab*4
volmintrab=10
volmaxtrab=50
ti_rc=1
tf_rc=4
ti_setup=1
tf_setup=3

volminent=volmintrab*ntrab
volmaxent=volmaxtrab*ntrab

tmin=int((ti_rc+ti_setup)*ntrab/2)
deltat=(hor-tmin)/nord
tmax=hor

timin=1
timax=int(hor/6)
tfmin=int(hor/2)
tfmax=hor

hmin=1
hmax=int(hor/10)
```

```
salida.write('int numtrab = '+str(ntrab)+';\n')
salida.write('int numord = '+str(nord)+';\n')
salida.write('int numproduc = '+str(nprod)+';\n')
salida.write('int numlin = '+str(nlin)+';\n')
salida.write('int nument = '+str(nent)+';\n')
salida.write('int horizon ='+str(hor)+';\n')
salida.write('range domain = 1..horizon;')
salida.write('\n\n')


salida.write('''range binary = 0..1;
range trab = 1..numtrab;
range ord = 1..numord;
range produc = 1..numproduc;
range lin = 1..numlin;
range ent = 1..nument;

int I[trab,ord]=0;\n''')

oo=1
aux=0

#I
for i in range(1,ntrab+1):
        if aux == trabxord:
                aux= 0
                oo=oo+1
        salida.write('I['+str(i)+','+str(oo)+'] = 1;\n')
        aux=aux+1

salida.write('\n\n')


# rc
salida.write('int rc[trab,lin]=2;\n')

for i in range(1,ntrab+1):
        for l in range(1,nlin+1):
                salida.write('rc['+str(i)+','+str(l)+'] = '+str(random.randint(ti_rc,tf_rc))+';\n')

salida.write('\n\n')

#Setup
salida.write('int setup[trab,trab,lin] = 1;\n')

for i in range(1,ntrab+1):
        for j in range(1,ntrab+1):
                for l in range(1,nlin+1):
```

```
                        salida.write('setup['+str(i)+','+str(j)+','+str(l)+']= '
                        +str(random.randint(ti_setup,tf_setup))+';\n')

salida.write('\n\n')

#tin
salida.write('int tin[trab,produc]=0;\n')

for i in range(1,ntrab+1):
        salida.write('tin['+str(i)+','+str(random.randint(1,nprod))+'] =1;\n')

salida.write('\n\n')


#tinto
salida.write('int tinto[ent,produc]=0;\n')
for i in range(1,nent+1):
        salida.write('tinto['+str(i)+','+str(random.randint(1,nprod))+'] =1;\n')


salida.write('\n\n')

#lt
salida.write('int lt[trab]=5;\n')
for i in range(1,ntrab+1):
        salida.write('lt['+str(i)+']='+str(random.randint(volmintrab,volmaxtrab))+';\n')

salida.write('\n\n')

#vol
salida.write('int vol[ent]=1000;\n')
for i in range(1,nent+1):
        salida.write('vol['+str(i)+']='+str(random.randint(volminent,volmaxent))+';\n')

salida.write('\n\n')

#t
salida.write('int t[ord]=1000;\n')
tt=tmin
for i in range(1,nord+1):
        salida.write('t['+str(i)+']='+str(min(tt,hor))+';\n')
        tt=tt+random.randint(1,2*deltat)

salida.write('\n\n')
#ti
salida.write('int ti[ent]=0;\n')
for i in range(1,nent+1):
        salida.write('ti['+str(i)+']='+str(random.randint(timin,timax))+';\n')


salida.write('\n\n')
#tf
```

```
salida.write('int tf[ent]=0;\n')
for i in range(1,nent+1):
        salida.write('tf['+str(i)+']='+str(random.randint(tfmin,tfmax))+';\n')


salida.write('\n\n')
#h
salida.write('int h[trab]=1;\n')
for i in range(1,ntrab+1):
        salida.write('h['+str(i)+']='+str(random.randint(hmin,hmax))+';\n')


salida.close()
```

Apéndice B

Código en *Comet* de AGUCP

```
////////////////////// Modelo ////////////////////////////

import cotfd;
Solver<CP> cp();


var<CP>{int} g[trab](cp,domain);
var<CP>{int} x[trab,lin](cp,binary);
var<CP>{int} y[trab,trab](cp,binary);
var<CP>{int} yj[trab,trab](cp,binary);
var<CP>{int} zb[trab,ent](cp,binary);
var<CP>{int} o[trab](cp,domain);
UniformDistribution dist(0..1);

int t0 = System.getCPUTime();

solve<cp>{

forall(n in trab) //(1)
   cp.post( sum(l in lin) x[n,l] == 1 );

forall(n in trab) // (2)
   cp.post( g[n] >= h[n] );

forall(n in trab) //(3)
   cp.post( o[n] == g[n]+ sum(l in lin) x[n,l]*rc[n,l] );

forall(n in trab, nn in trab : n != nn ) //(4)
   cp.post( y[n,nn] <= 1 + (g[n]-g[nn])/(horizon+1) );

forall(n in trab, nn in trab,l in lin : n != nn ) //(5)
```

```
   cp.post( y[n,nn] <= 1-x[n,l]+x[nn,l] );

forall(n in trab, nn in trab,l in lin : n != nn ) //(6)
   cp.post( y[n,nn] >= x[n,l]+x[nn,l]-2+(g[n]-g[nn])/(horizon+1) );

forall(n in trab, nn in trab, l in lin : n > nn ) //(7)
   cp.post( 1 >= x[n,l]+x[nn,l]-y[n,nn]-y[nn,n] );


forall(n in trab, nn in trab,l in lin : n != nn ) //(8)
   cp.post( o[nn] <=  2*horizon*(1-y[n,nn])+g[n]+2*horizon*(2-x[n,l]-x[nn,l])-setup[nn,n,l] );


forall(n in trab) //(9)
   cp.post( sum(e in ent) zb[n,e] == 1);

forall(n in trab,e in ent,p in produc) // (10)
   cp.post( tin[n,p]*zb[n,e] <= tinto[e,p] );

forall(n in trab,e in ent,p in produc) // (11)
   cp.post( (1-tin[n,p])*zb[n,e] <= 1-tinto[e,p] );

forall(n in trab,e in ent) // (12)
   cp.post( o[n] <= tf[e]*zb[n,e]+horizon*(1-zb[n,e]) );

forall(n in trab,e in ent) // (13)
   cp.post( ti[e]*zb[n,e] <= g[n] );

forall(e in ent) // (14)
   cp.post( sum(n in  trab) lt[n]*zb[n,e] <= vol[e] );

forall(k in ord,n in trab : I[n,k]==1) // (15)
   cp.post( o[n] <= t[k] ) ;


forall(n in trab, nn in trab : n != nn ) // (16)
   cp.post( yj[nn,n] <= y[nn,n] );

forall(n in trab, nn in trab, nnn in trab : n != nn  && n != nnn && nn != nnn)  // (17)
   cp.post( yj[n,nn] <= 2-y[n,nnn]-y[nnn,nn] );

forall(n in trab, nn in trab : n != nn ) // (18)
   cp.post( y[n,nn] <= sum( nnn in trab) yj[n,nnn] );

forall(n in trab) // (19)
   cp.post( yj[n,n]== 0);


}

//////////////////////////////// algoritmo ////////////////////////
```

```
using{


forall(n in trab)
   select(l in lin)
      try<cp> cp.post(x[n,l] == 1);| cp.post(x[n,l] == 0);

int m;
forall(n in trab){
   m=1;
    cout << g << endl;
    while(!g[n].bound())
       try<cp> cp.post( g[n] == m ) ;| {cp.post( g[n] > m ); m=m+1;}

    }

label(zb);
}



////////////////////////////////// Post Proceso ////////////////////////////



cout << g << endl;
//cout << o << endl;
cout << x << endl;
cout << zb << endl;
cout << yj  << endl;
cout << "time: " << System.getCPUTime() - t0 << endl;


ofstream out("trabajosd_plot_input.txt");
forall(n in trab, l in lin, k in ord, e in ent : x[n,l] == 1 && zb[n,e] == 1 && I[n,k] == 1)
out << "N" << n << " E" << e << " K" << k << " " << l << " " << g[n] << " " << o[n] << endl;



out.close();
```

Apéndice C

Código en *Python* de verificador.py

```
import random
import itertools
import sys
from operator import itemgetter, attrgetter
from time import clock, time


########## Datos creados al azar ####################

ntrab=100
trabxord=5
nord=ntrab/trabxord
nprod=3
nlin=2
nent=9

hor=ntrab*4
volmintrab=10
volmaxtrab=50
ti_rc=1
tf_rc=4
ti_setup=1
tf_setup=3

volminent=volmintrab*ntrab
volmaxent=volmaxtrab*ntrab
tmin=int((ti_rc+ti_setup)*ntrab/2)
deltat=(hor-tmin)/nord
tmax=hor


timin=1
timax=int(hor/6)
```

```
tfmin=int(hor/2)
tfmax=hor

hmin=1
hmax=int(hor/10)
oo=1
aux=0
salida=open('data.co','w')
salida.write('''range binary = 0..1;
range trab = 1..numtrab;
range ord = 1..numord;
range produc = 1..numproduc;
range lin = 1..numlin;
range ent = 1..nument;
int I[trab,ord]=0;\n''')

#I
I = [[0 for j in range(nord)] for i in range(ntrab)]
for i in range(1,ntrab+1):
        if aux == trabxord:
                aux= 0
                oo=oo+1
        I[i-1][oo-1] = 1
        salida.write('I['+str(i-1)+','+str(oo-1)+'] = 1;\n')
        aux=aux+1
salida.write('\n\n')

# rc
salida.write('int rc[trab,lin]=2;\n')
rc = [[2 for l in range(nlin)] for i in range(ntrab)]
for i in range(1,ntrab+1):

        for l in range(1,nlin+1):
                rc[i-1][l-1]=random.randint(ti_rc,tf_rc)
                salida.write('rc['+str(i-1)+','+str(l-1)+'] = '+str(rc[i-1][l-1])+';\n')

salida.write('\n\n')



#Setup
salida.write('int setup[trab,trab,lin] = 1;\n')
setup = [[[1 for l in range(nlin)] for j in range(ntrab)] for i in range(ntrab)]
for i in range(1,ntrab+1):
        for j in range(1,ntrab+1):
                for l in range(1,nlin+1):
                        setup[i-1][j-1][l-1] = random.randint(ti_setup,tf_setup)
                        salida.write('setup['+str(i-1)+','+str(j-1)+','+str(l-1)+']=
                        '+str(setup[i-1][j-1][l-1])+';\n')

salida.write('\n\n')
```

```
#tin
salida.write('int tin[trab,produc]=0;\n')
tin = [ 0 for i in range(ntrab)]

for i in range(1,ntrab+1):
        tin[i-1] = random.randint(1,nprod)-1
        salida.write('tin['+str(i-1)+','+str(tin[i-1])+'] =1;\n')
salida.write('\n\n')

#tinto
salida.write('int tinto[ent,produc]=0;\n')
tinto = [ 0 for i in range(nent)]
for i in range(1,nent+1):
        tinto[i-1]=random.randint(1,nprod)-1
        salida.write('tinto['+str(i-1)+','+str(tinto[i-1])+'] =1;\n')
salida.write('\n\n')



#lt
salida.write('int lt[trab]=5;\n')
lt = [ 5 for i in range(ntrab)]
for i in range(1,ntrab+1):
        lt[i-1] = random.randint(volmintrab,volmaxtrab)
        salida.write('lt['+str(i-1)+']='+str(lt[i-1])+';\n')
salida.write('\n\n')



#vol

salida.write('int vol[ent]=1000;\n')
vol = [ 1000 for i in range(nent)]
for i in range(1,nent+1):
        vol[i-1] = random.randint(volminent,volmaxent)
        salida.write('vol['+str(i-1)+']='+str(vol[i-1])+';\n')
salida.write('\n\n')

#t

salida.write('int t[ord]=1000;\n')
t = [ 1000  for i in range(nord)]
tt=tmin
for i in range(1,nord+1):
        t[i-1] = min(tt,hor)
        salida.write('t['+str(i-1)+']='+str(t[i-1])+';\n')
        tt=tt+random.randint(1,2*deltat)
salida.write('\n\n')



#ti
```

```
salida.write('int ti[ent]=0;\n')
ti = [ 0 for i in range(nent)]
for i in range(1,nent+1):
        ti[i-1] = random.randint(timin,timax)
        salida.write('ti['+str(i-1)+']='+str(ti[i-1])+';\n')
salida.write('\n\n')

#tf

salida.write('int tf[ent]=0;\n')
tf = [ 0 for i in range(nent)]
for i in range(1,nent+1):
        tf[i-1] = random.randint(tfmin,tfmax)
        salida.write('tf['+str(i-1)+']='+str(tf[i-1])+';\n')
salida.write('\n\n')



#h

salida.write('int h[trab]=1;\n')
h = [ 0 for i in range(ntrab)]
for i in range(1,ntrab+1):
        h[i-1] = random.randint(hmin,hmax)
        salida.write('h['+str(i-1)+']='+str(h[i-1])+';\n')

salida.close()



#ef   Entregas factibles para producto p
ef=[]

for p in range(nprod):
        aux=[]
        for ee in range(nent):
                if tinto[ee] == p:
                        aux.append(ee)
        ef.append(aux)


################## Definicion del Ordenador ##################


def ordena(b,l,g):

        orden=[]
        for linea in range(nlin):
                aux=[]
                for n in range(ntrab):
                        if (b[n] == 1 or b[n] == 3) and l[n] == linea:
```

```
                                aux.append([n,g[n]])
                aux=sorted(aux,key=itemgetter(1))
                aux2=[]
                for i in range(len(aux)):
                        aux2.append(aux[i][0])
                orden.append(aux2)
        return orden


################ Definicion del verificador  ##################



def verifica(b,l,g,e):
        if len(l) != ntrab:
        #       print "Error de Dimensiones en Vector l"
                return False
        if len(g) != ntrab:
        #       print "Error de Dimensiones en Vector g"
                return False
        if len(e) != ntrab:
        #       print "Error de Dimensiones en Vector e"
                return False
        o = [ g[i]+rc[i][l[i]] for i in range(ntrab)]

        for n in range(ntrab):
                if b[n] == 2 or b[n] == 3:
                        if tinto[e[n]] != tin[n]:
        #                       print "Error para tipo de producto trabajo "+str(n)+"
         y entrega "+ str(e[n])
                                return False
                        if g[n] < ti[e[n]]:
        #                       print "Error para tiempo inicial trabajo "+str(n)+"
        y entrega "+ str(e[n])
                                return False
                        if tf[e[n]] < o[n]:
        #                       print "Error para tiempo final trabajo "+str(n)+"
        y entrega "+ str(e[n])
                                return False


        for n in range(ntrab):
                if g[n] < h[n] and (b[n] == 1 or b[n] == 3):
        #               print "Error para insumos trabajo "+str(n)
                        return False

        for ee in range(nent):
                suma=0
                for n in range(ntrab):
                        if e[n]==ee and (b[n] == 2 or b[n] == 3):
                                suma=suma+lt[n]
```

```
                if suma > vol[ee]:
        #               print "Entrega "+str(ee)+" supera capacidad de litros"
                        return False

        for oo in range(nord):
                for n in range(ntrab):
                        if I[n][oo] == 1 and t[oo] < o[n] and (b[n] == 1 or b[n] == 3):
        #                       print "Trabajo "+str(n)+" fuera de plazo para la orden "+str(oo)
                                return False

        orden=ordena(b,l,g)
        for ll in range(nlin):
                for i in range(len(orden[ll])-1):
                        n=orden[ll][i]
                        nn=orden[ll][i+1]
                        if o[n] > g[nn]-setup[nn][n][l[n]]:

        #                       print "No se cumple tiempo de setup al pasar del trabajo "
        +str(n)+" al trabajo "+str(nn)+" en la linea "+str(ll)
                                return False
        return True




################## AGUCP++ ###################################



start = time()
b=[ 0 for i in range(ntrab)]
l=[ 0 for i in range(ntrab)]
g=[ 0 for i in range(ntrab)]
e=[ 0 for i in range(ntrab)]

emerlin=[ 0 for i in range(ntrab)]
for n in range(ntrab):
        aux=1000
        for ll in range(nlin):
                if rc[n][ll] < aux:
                        l[n]=ll
                        aux=rc[n][ll]
        emerlin[n]=range(nlin)
        emerlin[n].remove(l[n])
booleano=False
numtrabout=0
cock=False
for n in range(ntrab):
        print n
        m=1
        booleano=False
```

```
        while(not booleano):
            if m> hor and len(emerlin[n]) > 0:
                print "Buscando en otra linea para trabajo "+str(n)
                l[n]=emerlin[n].pop()
                m=1
            if m> hor and len(emerlin[n])== 0:
                print "No se puede colocar trabajo " +str(n)
                numtrabout=numtrabout+1
                b[n]=0
                break
            b[n]=3
            g[n]=m
            for ee in ef[tin[n]]:
                e[n]=ee
                cock=verifica(b,l,g,e)
                if cock:
                    booleano=True
                    break
            m=m+1


o = [ g[i]+rc[i][l[i]] for i in range(ntrab)]


############################   Post Proceso   ######################


print "Tiempo "+str(time() - start)
print "Numero de Trabajos no incluidos "+str(numtrabout)
resul=open("trabajosd_plot_input.txt","w")
for n in range(ntrab):
    for oo in range(nord):
        if I[n][oo] == 1 and b[n] > 0 :
            resul.write("N"+str(n)+" "+"E"+str(e[n])+" "+"K"+str(oo)+"
                        "+str(l[n]+1)+" "+str(g[n])+" "+str(o[n])+"\n")

resul.close
```

Apéndice D

Código en *GAMS* de main.gms

```
scalar starttime; starttime = jnow;
option limcol=0;
option limrow=0;
$onlisting
*desactiva listado de simbolos empleados
$onempty
*admite conjuntos vacios
option solprint=off;
$include "DATOS\CONJUNTOS.TXT"
$include "DATOS\PARAMETROS.TXT"
$include "DATOS\ESCALARES.TXT"
$include "DATOS\TABLAS.TXT"
$include "DATOS\setjobs.TXT"

*******************************     Pre Procesamiento     *************************

SCALARS
Cota    /0/
Min     /0/
CotaGeneral     /0/
NG                /1000/
VOLMIN     /1/
;

loop(K,   Cota=0
          loop(N,  if(I(k,n)=1,        Min=1000000000000000000000000
                                       loop(L,  if(Min>rc(l,n), Min=rc(l,n));
                                                Cota=Cota+Min;
                                           );
                         );
               );
           if(CotaGeneral<Cota,  CotaGeneral=Cota);
    );
```

```
*****************************8******    Modelo    ************************************
variable
F0     "Funcion Objetivo"



integer variables
g(n)    "Tiempo en el se empieza procesar el trabajo n"
o(n)    "Tiempo en el que se completa el tiempo n"
f(k)    "Tiempo en el cual se han completado todos los trabajos de la orden k"
u(k)    "Parte positiva de f(k)-t(k)"


binary variable
x(n,l)    "Toma el valor 1 ssi el trabajo n se procesa en la linea l"
y(n,nn)   "Toma el valor 1 ssi el trabajo n se procesa despues del trabajo n'
en la misma linea"
yj(n,nn) "Toma el valor 1 ssi el trabajo n se procesa justo despues
del trabajo n' en la misma linea"
zb(n,e)   "Toma el valor 1 ssi el trabajo n se procesa con la entrega e"
;

o.up(N) = 10000 ;
g.up(N) = 10000 ;
f.up(K) = 10000 ;
u.up(K) = 10000 ;

equations

ec0
ec1(K)
ec2(N)
ec2_5(N,L)                          "No disponibilidad de linea para trabajo"
ec3(N)
ec4(N,K)
ec5(N)
ec6(N,NN)
ec7(N,NN,L)
ec8(N,NN,L)
ec9(N,NN,L)
ec10(N,NN,L)                         "Impone tiempos de setup"
ec16(N)

ec17(N)                             "Obliga a que a cada trabajo se le asocie una y una sola entrega"
ec18(N,E,P)                          "Obliga a que el trabajo n sea del mismo tipo que la
entrega e  V1.0"
ec19(N,E,P)                          "Obliga a que el trabajo n sea del mismo tipo que la
entrega e  V2.0"
ec20(N,E)                           "El trabajo en su ventana de tiempo si pertece a la
```

```
entrega_e__V1.0"
ec21(N,E)                    "El_trabajo_en_su_ventana_de_tiempo_si_pertece_a_la
entrega_e__V2.0"
ec22(E)                      "Capacidad_maxima_de_cada_entrega"

ec23(K)                      "Obliga_a_que_los_tiempos_de_las_ordenes_se_cumplan_estrictamente"




ec27(N,L,E)                  "Impone_trabajos_previos_dados"
ec28(N,L,E)                  "Impone_trabajos_previos_dados"
ec29(N,L,E)                  "Impone_trabajos_previos_dados"

ec37(N,NN)                  "Definicion_de_yj_1.0"
ec38(N,NN,NNN)              "Definicion_de_yj_2.0"
ec39(N,NN)                  "Definicion_de_yj_3.0"
ec40(N)                     "Definicion_de_yj_4.0"



;


ec0..                                          F0 =e=    sum(n,o(n)) ;
ec1(K)..                                       u(k) =g= f(k)-t(k);
ec2(N)..                                       sum(l,x(n,l)) =e= 1;
ec2_5(N,L)$(rc(l,n) = -1 )..                   x(n,l) =e= 0;
ec3(N)..                                       g(n) =g= h(n);
ec4(N,K)$(I(k,n)=1)..                          f(k) =g= o(n);
ec5(N)..                                       o(n)=e= g(n)+sum(l,x(n,l)*rc(l,n));
ec6(N,NN)$(ord(n) ne ord(nn))..                y(n,nn) =l= 1+(g(n)-g(nn))/(M+1);
ec7(N,NN,L)$(ord(n) ne ord(nn))..              y(n,nn) =l= 1+x(n,l)-x(nn,l);
ec8(N,NN,L)$(ord(n) ne ord(nn))..              y(n,nn) =g=  x(n,l)+x(nn,l)-2+(g(n)-g(nn))/(M+1);
ec9(N,NN,L)$(ord(n) gt ord(nn))..              1 =g= x(n,l)+x(nn,l)-y(n,nn)-y(nn,n);

ec10(N,NN,L)$(ord(n) ne ord(nn))..             o(nn)+setup(nn,n,l) =l= 2*M*(1-y(n,nn))
+g(n)+2*M*(2-x(n,l)-x(nn,l));

ec16(N)..                                      g(n) =l= M;


ec17(N)..                                      sum(e, zb(n,e))=e= 1;
ec18(N,E,P)..                                    tin(n,p)*zb(n,e) =l= tinto(e,p);
ec19(N,E,P)..                                    (1-tin(n,p))*zb(n,e) =l= 1-tinto(e,p);
ec20(N,E)..                                    o(n) =l= tf(e)*zb(n,e)+M*(1-zb(n,e));
ec21(N,E)..                                    ti(e)*zb(n,e) =l= g(n);
ec22(E)..                                      sum(n,lt(n)*zb(n,e)) =l= vol(e);

ec23(K)..                                      f(k)  =l= t(k) ;
```

```
ec27(N,L,E)$(sjb(n,l,e) gt 0 )..                    x(n,l) =e= 1;
ec28(N,L,E)$(sjb(n,l,e) gt 0 )..                    zb(n,e) =e= 1;
ec29(N,L,E)$(sjb(n,l,e) gt 0 )..                   g(n) =e= sjb(n,l,e);

ec35(N)..                                            g(n) =g= 1;

ec37(N,NN)$(ord(n) ne ord(nn))..       yj(n,nn) =l= y(n,nn);
ec38(N,NN,NNN)$(ord(n) ne ord(nn)  and (ord(n) ne ord(nnn)) and (ord(nn) ne ord(nnn)))..
yj(n,nn) =l= 2-y(n,nnn)-y(nnn,nn);
ec39(N,NN)$(ord(n) ne ord(nn))..        y(n,nn)  =l= sum(nnn,yj(n,nnn));
ec40(N)..      yj(n,n)  =e= 0;
```

```
model SanPedro /all/     ;

SanPedro.iterlim=1200000000;
SanPedro.reslim=TIME_LIMITE;
SanPedro.optcr=GAP;
SanPedro.optfile=1;

option mip = cplex;
$onecho>cplex.opt
threads 4
depind 1
mipstart 1
$offecho

solve SanPedro using mip minimixing F0;

display g.l , o.l , f.l , x.l,zb.l, y.l ;
```

```
*********************************     Post Procesamiento     ************************************

file  SolGams /SolGams.TXT/;
put SolGams;
```

```
SolGams.pw = 32000

put 'TRABAJO____ENTREGAS______LINEA_____TI_____TF_____'/;
loop((N,L,E),

     if(x.l(n,l)=1  and  zb.l(n,e)=1,
          put  put n.tl '_______' (ord(e)-1):0:0  '_______' (ord(l)-1) :0:0 '______'
          g.l(n) :0:0
           '______' o.l(n):0:0  '______'    /;

        );


     );
putclose SolGams
```