UNIVERSIDAD DEL VALLE

Implementación de un prototipo para solucionar el problema de *Job-Shop Scheduling* utilizando Programación Concurrente con Restricciones

por

Francisco Javier Peña Escobar

Tesis profesional presentada como requisito parcial para obtener el título de Ingeniero en Sistemas y Computación

en la

Facultad de Ingeniería
Escuela de Ingeniería de Sistemas y Computación
AVISPA

23 de septiembre de 2008

"Si he visto más allá que otros, es por pararme sobre los hombros de gigantes."

Isaac Newton

UNIVERSIDAD DEL VALLE

Resumen

Facultad de Ingeniería
Escuela de Ingeniería de Sistemas y Computación
AVISPA

Ingeniero en Sistemas y Computación

por Francisco Javier Peña Escobar

En un problema de *scheduling*, existen un conjunto de restricciones (sobre las tareas, objetos del problema, uso de recursos compartidos, entre otros), que deben satisfacerse para poder obtener una solución.

En este trabajo de grado se implementa un modelo de resolución para problemas típicos de *scheduling*, que pueden especificarse como un problema de satisfacción de restricciones temporales métrico-disyuntivas entre puntos del tiempo. El modelo y el motor de soluciones fueron implementados en el lenguaje de programación *Mozart* conjuntamente con una interfaz gráfica desarrollada en *Java* que se encarga de invocarlos. Para poder llevar esto a cabo, se hizo uso de un marco de representación para especificar problemas de scheduling mediante un conjunto de restricciones temporales métrico-disyuntivas basadas en puntos de tiempo.

Por último, se compararon los tiempos de respuesta de la aplicación final con tiempos de respuesta de problemas típicos de la literatura abordados desde otros paradigamas de resolución.

Este trabajo de grado está basado en la tesis doctoral "Un modelo de integración de técnicas de CLAUSURA y CSP de restricciones temporales: Aplicación a problemas de *Scheduling*" propuesta por María Isabel Alfonso Galipienso [1].

Agradecimientos

Quiero agradecer a mi director de Tesis, Juan Francisco Días, por su continuo apoyo. A mi codirector de Tesis, Héctor Hernán Toro por sugerirme este problema como proyecto de grado y por el material que me brindo.

A todos en la Escuela de Ingeniería de Sistemas y Computación, especialemente a los profesores por sus enseñanzas para formarme como profesional. A la Universidad del Valle, por darme la oportunidad de estudiar y las herramientas necesarias para hacerlo. Al grupo Avispa.

A mis padres y a mi hermana por su continuo apoyo durante toda mi carrera, muchas muchas gracias.

Finalmente a mis amigos y a todos aquellos que hayan contribuído en mi formación como profesional.

Índice general

Resumen II

Agradecimientos III

Índice de figuras VI

Índice de cuadros VII

1. El problema de *scheduling* 1
- 1.1. Conceptos básicos sobre scheduling 1
- 1.2. Tipología de problemas de scheduling 4

2. Programación concurrente con restricciones 6
- 2.1. Nociones. 6
- 2.2. Problemas que utilizan *CCP*. 7
- 2.3. Representaciones de un *CSP* 7
- 2.4. Clases de restricciones. 8
- 2.5. Agentes Tell y Ask. 9
- 2.6. Propagación. 10
- 2.7. Exploración. 10
- 2.8. Consistencia. 11
- 2.9. Restricciones tratables. 11
- 2.10. Precompilación de problemas. 12
- 2.11. Limitaciones. 12
- 2.12. El problema max - CSP. 13

3. Modelo de resolución 14
- 3.1. Elementos temporales de *scheduling* 14

3.2. Representación de las operaciones 15
3.3. Restricciones no disyuntivas 16
3.3.1. Tiempos de preparación de las máquinas 18
3.4. Restricciones disyuntivas 19
3.4.1. Patrón de flujo 19

4. Implementación del modelo usando CCP **23**
4.1. Archivo de entrada 23
4.2. Estructura de datos 24
4.3. Restricciones no-disyuntivas 24
4.4. Restricciones disyuntivas 25
4.5. Estrategias de propagación 26
4.6. Estrategias de distribución 26
4.7. Búsqueda de soluciones 27

5. Interfaz de la aplicación **28**

6. Experimentación y pruebas **31**
6.1. El problema *ft10* 31
6.2. Otros problemas 33

7. Conclusiones y trabajo futuro **34**
7.1. Según los resultados obtenidos en el Capítulo 6 de Experimentación y pruebas 34
7.2. Conclusiones generales 35
7.3. Trabajo futuro 35

Bibliografía **37**

Índice de figuras

3.1. Representación gráfica de una actividad 16
3.2. Restricciones temporales asociadas a un trabajo J_i 17
3.3. Representación de las operaciones mediante un único punto temporal 18
3.4. Tiempos de preparación 19
3.5. Restricción disyuntiva 20
3.6. Representación de *flow-shop*. 21
3.7. Representación de *job-shop*. 22

5.1. Interfaz inicial 29
5.2. Interfaz desplegando una solución 30

Índice de cuadros

6.1. Paradigmas de otras implementaciones 32
6.2. Comparación con *ft10* 32
6.3. Comparación con otras instancias 33

A mis padres

Capítulo 1

El problema de *scheduling*

El resumen a continuación fue tomado de [1].

Un problema de *scheduling* se define como el proceso de asignar *recuros* a *actividades* a lo largo del tiempo [2] o, alternativamente, la determinación de cuándo las operaciones o acciones que componen un plan deben ser realizadas y hacer uso de dichos recursos. Adicionalmente, hay que garantizar que los tiempos asignados a dichas acciones deben cumplir con una serie de restricciones establecidas en el plan, así como una optimización de determinados criterios [3]. De esta forma, *scheduling* está directamente asociado con la ejecutabilidad y optimalidad de un plan preestablecido. El proceso de *scheduling* es particularmente importante en el campo de la producción y de la gestión de operaciones, y así su terminología más relevante deriva de esta fuente como por ejemplo *trabajos*, *recursos* o *actividades*.

A continuación se presenta una descripción de algunos conceptos relacionados con *scheduling*.

1.1. Conceptos básicos sobre scheduling

En general, en un problema de *scheduling* intervienen los siguientes elementos:

- Trabajos
- Actividades
- Máquinas
- Tipos de *scheduling* (patrón de flujo)
- Objetivos

Cualquier problema de *scheduling* consta de uno o más **trabajos**. Un trabajo es el término usado para designar un único elemento o una serie de elementos que tienen que ser procesados en los distintos recursos. Normalmente se identifica con un producto a entregar que tiene que alcanzar una cierta calidad. Los trabajos, en la mayoría de los casos, tienen restrigido su tiempo de comienzo (*ready time*: tiempo a partir de cual puede comenzar la ejecución del trabajo).

Cada trabajo, a su vez está compuesto de una o mas **actividades**. Se denomina actividad (u operación) al procesamiento de un trabajo particular sobre un recurso particular, y por lo tanto constituyen las unidades elementales de procesamiento. Las actividades pertenecientes a un mismo trabajo guardan un orden de precedencia conocido a priori (*restricciones de precedencia o tecnológicas*), que debe respetarse, pudiendo haber variaciones en cuanto a posibles solapes entre las actividades de un mismo trabajo, o estar separadas por intervalos de tiempo amplios. El tiempo de procesamiento de cada actividad o duración puede ser fija, variable en un intervalo o estar formada por un conjunto de valores/intervalos disjuntos. Además, en un problema de *scheduling* puede haber actividades de distintos tipos, que condicionarán su procesamiento en un determinado tipo de máquina. A lo largo de este trabajo se hará uso de los términos actividad u operación indistintamente.

Cada actividad se procesa (o es consumida) por una **máquina** (o recurso), o por un tipo de máquina, si se trata de un problema de asignación de recursos. Normalmente se asume que una máquina únicamente puede realizar una actividad o tarea a la vez, es decir no hay solapamiento entre actividades que se procesan en una misma máquina, lo cual da lugar a las llamadas *restricciones disyuntivas o de capacidad*. A lo largo de trabajo se hará uso indistintamente de los términos máquina o recurso.

Es importante conocer el **tipo de *scheduling*** que se quiere realizar, lo cual definirá el patrón de flujo del problema, o lo que es lo mismo, la secuencia de utilización de las máquinas por parte de las actividades, en caso de que se trate de un problema de *scheduling* puro (ver apartado 1.2).

Finalmente, dependiendo de los **objetivos** perseguidos por el usuario, podemos distinguir los siguiente conceptos:

- *Factibilidad*, relativo a la satisfacción del conjunto de restricciones preestablecidas en un determinado plan. Estas restricciones pueden, típicamente, corresponder a secuncialidad de las acciones, capacidad de los recursos y duración o requerimientos de uso de los recursos.
- *Optimalidad*, relativo a la optimización de una o varias funciones objetivo, dependientes del entorno de aplicación o del criterio del usuario. Cuando se persigue la optimalidad, la complejidad de los algoritmos de resolución aumenta, debido a la mayor dificultad del proceso de búsqueda asociado. Uno de los criterios de optimización más usual es el de la minimización del máximo tiempo de finalización del *scheduling* (*makespan*).

Los elementos básicos presentados hasta el momento pueden representarse como un conjunto de restricciones temporales métrico-disyuntivas formado por:

- Duración de las actividades *("la actividad X tiene un duración de Y unidades de tiempo").*
- Restricciones de precedencia para las actividades de cada trabajo *("El trabajo J está formado por la siguiente sucesión de actividades: primero la actividad A, después la B, ...").*
- Restricciones de inicio y terminación para cada trabajo *("el trabajo J tiene que comenzar a partir del momento P, y terminar antes del momento Q").*
- Restricciones de capacidad sobre el uso de recursos *("la actividad X se ejecuta antes o después de la actividad Y, puesto que ambas utilizan la misma máquina").*

En el apartdado siguiente, se expondrán algunos de los tipos de problemas de *scheduling* existentes, en función de las restricciones asociadas al mismo.

1.2. Tipología de problemas de scheduling

Se pueden hacer muchas clasificaciones de los problemas de *scheduling*. Por ejemplo, se podrían caracterizar en función de parḿetros como: determinismo frente a estocasticidad (duraciones de las tareas fijas o probabilistas); requerimientos de tiempo real (restricciones de tiempo real estricto frente a procesos *off-line*); presencia de restricciones sobre los recursos; objetivos del *scheduling* (minimizar el *makespan, el coste,* ...); requerimientos de optimalidad frente a satisfacibilidad; etc.

Independientemente de los parámetros mencionados, existen dos grandes familias de *scheduling* según los grados de libertad en cuanto a la demanda de recursos en un cierto tiempo y el suministro de dichos recursos:

- Problemas de *scheduling* "puros"
- Problemas de asignación de recursos

En los problemas de ***scheduling* "puros"**, la capacidad de cada recurso está definida sobre un cierto número de intervalos temporales y el problema consiste en cubrir las demandas de recursos de las actividades a lo largo del tiempo, sin exceder sus capacidades disponibles. En este tipo de problemas se conoce a priori qué recurso va a utilizar exactamente cada una de las actividades. Dependiendo del uso de los recursos por parte de las actividades, podemos distinguir cuatro *patrones de flujo*:

(a) Open-Shop: No existe ninguna restricción en cuanto al orden de uso de los recursos por las actividades de cada uno de los trabajos.

(b) Job-Shop: Cada trabajo, o conjunto de actividades, debe usar los recursos en un orden determinado (*restricciones tecnológicas*). "N" trabajos deben ser procesados, una sola vez por "M" recursos, con un orden y durante un tiempo dado.

(c) Flow-Shop: Todos los trabajos utilizan los recursos en el mismo orden. Es un caso particular del job-shop.

(d) Permutación Flow-Shop: Todos los trabajos utilizan los recursos en el mismo orden. Todos los recurson procesan los trabajos en el mismo orden. Es un caso particular del flow-shop.

Además, se puede imponer la restricción adicional de que cada una de las actividades pueda interrumpirse y ser continuada más tarde, después de que el recurso le vuelva a ser asignado. En este caso, se está hablando de *scheduling* con reemplazo (*preemptive scheduling*). Si la ejecución de cada actividad tiene que realizarse en su totalidad antes de abandonar el recuros asignado, se está ante un tipo de *scheduling* denominado sin reemplazo (*non-preemptive scheduling*).

En este trabajo únicamente se van a tener en cuenta instancias de tipo *job-shop* y *flow-shop*. Además se considerarán instancias sin reemplazo.

En los problemas de **asignación de recursos**, disponemos de un conjunto de operaciones, y para cada una de ellas se dispone de un conjunto de recursos idénticos (realizan el mismo tipo de operación), pero no equivalentes (pueden requerir distintos tiempos o costes de procesamiento), que pueden ser utilizados. Por ejemplo *una determinada operación o_i consistente en la impresión de un documento puede realizarse indistintamente en la máquina m_p (que es una impresión láser); si se asigna a la máquina m_p tendrá ima duración de 10 minutos, mientras que si se utiliza m_q su duración será de 5 minutos. El coste de utilización de las máquinas es de 1000pts/hora.* En este caso no se conoce a priori qué recurso concreto va a utilizar cada operación, sino que el problema va a consistir en asignar los recursos a tiempo para garantizar que se cumplen todas las demandas. El problema de la asignación de recursos no va a ser abordado en esta tesis.

El tamaño de un problema de *scheduling* varía desde un número reducido de actividades a miles de ellas. Otras características numéricas importantes varían de un entorno a otro. Por ejemplo, la variación de la duración de las operaciones de fabricación depende de los productos a fabricar, y la importancia de los recursos que representan “cuellos de botella” varía con la carga global de la fábrica.

Capítulo 2

Programación concurrente con restricciones

El resumen a continuación fue tomado de [4].

2.1. Nociones.

La expresión "programación concurrente con restricciones" integra tres nociones fundamentales. Primero "programación" se refiere a programación de un computador. Un programa de computador es una expresión de un método computacional de un lenguaje de computador. Segundo, "concurrente" se refiere a que el método computacional ofrece la posibilidad de establecer ordenamientos mínimos de las acciones de modo que varias de ellas puedan eventualmente realizarse simultáneamente. Finalmente "restricciones" hace referencia a que las acciones se expresan mediante condiciones que debe cumplir la solución de un problema para que se acepte como factible. La programación por restricciones es entonces una metodología particular para programar computadores.

2.2. Problemas que utilizan *CCP*.

Hay dos clases de problemas para los que se estima conveniente utilizar programación por restriciones: Problemas de satisfacción de restricciones y problemas de optmización combinatoria. En los problemas de satisfaccón de restricciones *(CSP)*, se busca obtener soluciones factibles. En los problemas de optimización combinatoria, se busca además que la solución factible conduzca a la minimización (o maximización) de una función objetivo. La estructura particular de estos dos tipos de problemas, que se detallará mas adelante, se adapta a la metodología empleada por *CCP* para resolverlos: Representar un modelo del problema en un lenguaje de computador y describir una estrategia de búsqueda para resolverlo. Para la representación es conveniente que el lenguaje de computador es cuestión permita describir el modelo en términos cercanos al área de aplicación. La estrategia de búsqueda, por otra parte, debe conducir a encontrar soluciones de manera eficaz.

2.3. Representaciones de un *CSP* .

En un *CSP* se trata de encontrar una solución factible, sujeta a una serie de condiciones o restricciones sobre el conjunto de valores de las variables. Usualmente se requiere que las restricciones, expresadas en el lenguaje de programación, sean fáciles de evaluar. En general son formas matemáticas cerradas. Los valores de las variables toman usualmente valores dobre un conjunto discreto. En los problemas de optimización se requiere así mismo que la función objetivo sea fácil de evaluar. Una representación de un *CSP* puede ser la siguiente:

- Variables: $X_1,\ X2,\ \ldots,\ Xn$
- Valores posibles (dominios): $D_i,\ D_2,\ \ldots,\ D_n$, donde D_j es el dominio de Xj. El dominio puede ser finito o infinito.
- Las restricciones son funciones (predicados, realmente): $f(X_1, \ldots, X_n) \in \{0,1\}$. Cuando el valor de la función es uno, se dice que los valores en cuestión satisfacen la restricción.

2.4. Clases de restricciones.

Concretamente, en un *CSP* se trata de encontrar valores de $X_1, \ldots, X_n$ tales que: $X_j \in D_j$, para todo $j = 1, 2, \ldots, n$ $f_k(X_1, \ldots, X_n) = 1$, para $k = 1, 2, \ldots, m$. Los dominios pueden ser cualquier conjunto. Por ejemplo los enteros *1 ... 100*, o los nombres {*Pedro, Juan, María*}, o los reales en el intervalo *[0, 50]*. Simplemente, las restricciones pueden ser de muchas clases. Por ejemplo:

- Restricciones lógicas: si *Caudal* $>$ *67,5* entonces *Apertura* $\neq$ *0*. La actividad A debe preceder a la actividad B, o bien la actividad B debe preceder a la actividad A.
- Restricciones globales: Todos los valores de $X_1, X_2, \ldots, X_n$ deben ser diferentes.
- Cardinalidad: El i-ésimo elemento del arreglo *SALONES* es el número de veces qye el i-ésimo elemento del arreglo *TIPOS* aparece en el arreglo *ASIGNACIONES*.
- Meta restricciones: El número de veces que 5 aparece en el arreglo A es exactamente igual a 3.
- Restricciones de elemento: El costo de asignar a la persona i la tarea j es *costo[tarea[i]]*, cuado *tarea[i]* $=$ j.

En este contexto, la programación concurrente por restricciones provee:

- Una metodología de modelamiento para identificar convenientemente las variables, las restricciones y la función objetivo (si la hay) de un problema.
- Un lenguaje de programación (*Mozart*, en nuestro caso), para expresar algoritmos de búsqueda que encuentran valores de las variables que satisfagan todas la retricciones y optimicen el objetivo.
- Un *sistema de programación* que incluya: Restricciones predefinidas con algoritmos poderosos de *filtrado* o *propagación* de valores. que reduzcan de manera efectiva el tamaño del espacio de búsqueda. Funcionalidad para la definición de nuevas restricciones y algoritmos de filtrado.

En los lenguajes *CCP*, las restricciones constituyen entonces componentes básicos, que expresan información *parcial* sobre las variables. Cada restricción está a cargo de un *agente*, cuyo papel es velar por el cumplimiento de esa restricción, propagando a los otros agentes sus efectos (esto es, la nueva información que se deduzca de ella). Todos estos agentes operan de manera concurrente.

2.5. Agentes Tell y Ask.

La memoria contiene predicados que representan información parcial sobre las variables y los agentes interactúan con la memoria agregando información (agentes *tell*) o haciendo preguntas (agentes *ask*). Los *tell* agregan una restricción a la memoria y los *ask* preguntan si, de las restricciones ya contenidas en la memoria, se puede deducir una restricción dada.

Leyendo los agentes en el sentido de las manecillas del reloj, se agrega primero la información $X > 10$ y luego se pregunta si $X < 50$ es cierto. Como esto no se puede afirmar, el agente *ask* se bloquea. Similarmente, el agente $ask(X = 15) \rightarrow Q$ se bloquea. Cuando el agente $tell(X < 20)$ agregue su información, el $ask(X < 50) \rightarrow P$ podrá "despertar" y continuar con el proceso P. El resultado de un cálculo es, en este modelo, el conjunto de valores de las variables compatible con la totalidad de las restricciones que se ha agregado a la memoria.

A diferencia de los lenguajes tradicionales, la memoria denota aquí información *parcial*. Cada vez que se impone una restriccón mediante la operación *tell*, hay un proceso de propagación de sus efectos que consiste, en términos generales, en identificar restricciones adicionales que son su consecuencia lógica. Es perfectamente posible que el conjunto de restricciones así obtenido no determine de manera única los valores de las variables. En este caso se hace necesario un proceso de *exploración*, básicamente una estrategia de ensayo y error inteligente, que busca valores precisos, compatibles con la información parcial en la memoria.

Todos los lenguajes de programación por restricciones garantizan eficiencia en el proceso de propagación. Se distinguen, sin embargo, en el número y tipo de restricciones que propagan. Un tipo usual es el llamado de *restricciones de dominio finito*, En éste, se supone que las variables tienen un conjunto finito de valores posibles, usualmente de números enteros.

Ninguno de los lenguajes puede siempre garantizar la eficiencia en el proceso de exploración, debido a la característica de NP-completitud de la mayoría de los problemas *CSP* (esto es, solamente se conocen algoritmos cuyo tiempo de ejecución crece exponencialmente con la cantidad de variables). Lo que se busca es un balance adecuado entre propagadores poderosos y técnicas inteligentes de exploración que conduzcan a una resolución eficaz de los problemas en muchos casos.

2.6. Propagación.

Las estrategias de propagación o filtrado de valores utilizan técnicas muy eficientes (y, por consiguiente, incompletas) que razonan sobre los límites de los dominios. Por ejemplo si $X \in$ *[3, 7]*, $Y \in$ *[2, 6]* y $Z \in$ *[1, 9]*, el agente encargado de la restricción $X + Y < Z$ procederá a eliminar el valor 7 para X, luego el valor 6 para Y, y después los valores 1, 2, 3, 4 para Z. Los dominios resultantes $X \in$ *[3, 6]*, $T \in$ *[2, 5]*, $Z \in$ *[5, 8]* contienen las soluciones posibles, aunque hay combinaciones de valores que no son factibles. Las reducciones de los dominios de las variables pueden resultar a través de otras restricciones, en reducciones de dominios de otras variables, que pueden ocasionar a su vez reducciones adicionales de X, Y, Z.

2.7. Exploración.

Cuando este proceso de propagación se estabiliza, interviene la etapa de exploración. La exploración consisten en seleccionar algún valor del dominio resultante de alguna variable y nuevamente propagar sus efectos. El método que se emplea comúnmente es alguna variante de *foward-checking*. Lo interesante de los lenguajes *CCP* es que la escogencia de un valor se modela como la imposición de una nueva restricción y puede por tanto involucrar estrategias sofisticadas de particionamiento del dominio de la variable seleccionada. Se podría, por ejemplo, partir dominio de X en tres, mediante las restricciones $X < 5$, $X = 5$ y $X > 5$. El proceso de exploración escoge alguna de estas alternativas, impone la restricción correspondiente y propaga sus efectos. Si se escoge la tercera, por ejemplo, inmediatamente se deduce la solución $X = 6,\ Y = 2,\ Z = 8$.

2.8. Consistencia.

En la etapa de propagación, se utilizan técnicas llamadas de *consistencia*. Razonar sobre los límites de los dominios (proceso que se denomina *límite-consistencia*), es solamente una de las alternativas posibles. La propagación por *arco-consistencia*, por ejemplo, recorre exhaustivamente todos los valores de cada dominio buscando eliminar aquellos que no tengan un *soporte* en otro dominio. Es decir, para cada valor de un dominio, se busca al menos un valor en los otros que sirva de testigo sobre la validez de cada restricción. Por ejemplo, si agregamos la restricción *si* $X = 5$ *entonces* $Y > 3$, *límite-consistencia* no eliminará los valores adicionales. El proceso de *arco-consitencia*, por el contrario, elimina el valor 5 de X. El proceso de *arco-consistencia* es efectivo en muchos casos y existen además algoritmos eficientes para realizarlo.

Los algoritmos de *arco-consistencia* logran que cada valor de un dominio tenga al menos un soporte en los otros dominios. Esta misma idea prodría extenderse a *parejas* de valores. Es lo que se llama *camino-consistencia* o *2-consistencia*. El interés es el de filtrar aún mas los dominios. Obviamente, a medida que se incrementa el grado de consitencia, aumenta también la complejidad de los algoritmos. Se debe entonces encontrar el balance justo entre el tiempo que es dable a dedicar a la reducción de dominios (propagación) y el que se dedica posteriormente a la exploración de los dominios reducidos. Diferentes alternativas de propagación, que se sitúan en el intermedio entre *arco-consistencia* y *camino-consistencia*, se han propuesto recientemente.

2.9. Restricciones tratables.

Una alternativa complementaria a reducción de dominios es la de caracterizar tipos de restricciones *tratables*. Estas son restricciones para las cuales es demostrable que existen algoritmos eficientes para decidir si hay o no una solución que las satisfaga. El ideal sería entonces tener disponibles un conjunto de tales restricciones, y construír las que sean relevantes para un problema dado mediante combinaciones (disyunciones, por ejemplo) simples de ellas. De esta forma, en principio, podrían utilizarse los algoritmos eficientes para resolver por completo el problema, al menos para un subconjunto de las restricciones. Se

han identificado recientemente clases interesantes de disyunciones de restricciones que son tratables. Resta investigar sobre los mejores mecanismos para integrar soluciones totales o parciales de subproblemas definidos por restricciones tratables con las demás restricciones, de manera que se provean herramientas efectivas para identificar posibles inconsistencias en la definición de un problema.

2.10. Precompilación de problemas.

Los *CSPs* que ocurren en la realidad son problemas complejos que involucran un número elevado de variables y/o dominios grandes. Esto impone la utilización de estrategias de uso eficiente del espacio en memoria tanto para la propagación como durante la exploración. Una metodología que puede resultar efectiva es la de pre-compilación de subproblemas para los que es viable pre-compular soluciones, utilizando estructuras de datos eficientes para representar esas soluciones parciales.

2.11. Limitaciones.

Las tecnologás *CCP* tienen por el momento una importante limitación en cuanto a su potencial de utilización generalizada, fruto de la escasa información que proveen los mecanismos de solución a problemas. Cuando las restricciones que modelan un problema son incosistentes, al usuario le interesa no solamente saberlo sino también las razones precisas de esa inconsistencia. Por ejemplo, las variables, dominios o restricciones que contribuyen realmente a la inconsistencia. Hay muy poco trabajo desarrollado en esta dirección, quizá por la inherente complejidad del problema. Una posible estrategia viene de la literatura de revisión de conocimento en Inteligencia Artificial, que sugiere algunas metodologías que podrían explorarse para su integración en el contexto de *CCP*. Por otro lado, las técnicas de registro de valores incompatibles (*nogoods*) para problemas *CSP* dinámicos pueden dar ideas sobre estrategias para representar las causas de posibles fallas en la exploración.

2.12. El problema max - CSP.

Buena parte de los problemas que ocurren en la práctica están sobrerrestringidos (es decir, son inconsistentes), no por errores en la especificación, sino porque es muy difícil o imposible conocer de antemano el mínimo conjunto de restricciones que modela de manera completa el problema. Se trata entonces de encontrar el máximo número de restricciones que se puede satisfacer de modo que el problema tenga solución. Es lo que se llama el problema *max-CSP*. En estos casos se asume que las restricciones son *preferencias* que es conveniente satisfacer, si se puede. A este tipo de restricciones se les denomina *débiles*. Existen actualmente varias propuestas para extender la noción de consistencia a restricciones débiles. Estos métodos, con mayor o menor grado de filtraje de valores, son adecuados dentro del contexto de *CSP*, pero plantean dificultades en el marco de *CCP*, debido fundamentalmente a que en éste los mecanismos de computación son inferencias lógicas. En una inferencia lógica se impone la validez o invalidez de una restricción, pero no su *preferencia*. Solamente hasta hace muy poco se ha propuesto un modelo para la integración de restricciones débiles en el marco de los *CCP*.

Capítulo 3

Modelo de resolución

Parte del éxito a la hora de implementar un modelo de satisfacción de restricciones está en representar clara y consistentemente la información que nos brinda el problema. En este capítulo se describe el modelo planteado en [1] (fue tomado textualmente).

3.1. Elementos temporales de *scheduling*

La representación temporal a seguir contiene una mayor expresividad que aproximaciones anteriores, de forma que se pueden representar restricciones de *scheduling* no contempladas en otros modelos de resolución, como por ejemplo la consideración de los tiempos de mantenimiento de los recursos, periodos de descanso, consideración de los costes de uso de los recursos o fabricación por lotes. [1]

Esta representación tiene como base objetos temporales primitivos, que le permiten representar distintos elementos del sistema. Se hace uso de los **puntos temporales**, considerando a ***T0*** como el origen del tiempo.

También se toma en consideración que el tiempo es una sucesión discreta de puntos temporales, lo cual es común en la mayor parte de problemas de *scheduling*, y dependiendo de la instancia en particular se puede utilizar una **granularidad** adecuada.

La información temporal con la que se va a tratar es **métrica**, es decir, cuantitativa, y en la que una actividad está representada por dos puntos de tiempo (t_i, t_j) los cuales marcan su inicio y fin, respectivamente. De esta forma podemos representar expresiones como *"la actividad A tiene una duración de 8 horas", "la actividad A puede empezar 2 minutos a partir del inicio del* scheduling*"*; aunque también se puede expresar información que esté en forma cualitativa, por ejemplo: *"La actividad F debe empezar antes que la actividad B"* puede verse como *"Hay un intervalo de tiempo comprendido de* $(0, \infty)$ *entre el final de la actividad F y el inicio de la actividad B"*.

Además la información puede ser disyuntiva, de forma tal que podemos tratar expresiones de tipo: *"la actividad B tiene lugar 3 ó 5 semanas después de la actividad C", "la actividad X se ejecuta 10 minutos antes o entre 10 y 20 minutos después de la actividad C"*.

3.2. Representación de las operaciones

Las operaciones constituyen unidades de procesamiento elementales. Cada una de ellas se realiza sobre una máquina (recurso) de manera indivisible, es decir, no interrumpible.

Podemos representar una operación o mediante la siguiente información:

- t_i: punto temporal que indica el inicio de una operación
- t_j: punto temporal que indica el final de una operación
- d: duración de la operación

Gráficamente los puntos temporales t_i y t_j serán dos nodos en una red de restricciones, como se puede ver en la Figura 3.1. La arista que une los nodos t_i y t_j representa el espacio de tiempo transcurrido entre el inicio de la operación, y el final de la misma, que en este caso se corresponde con la duración d de la operación.

Llamaremos ***on(o)*** al punto temporal que representa el inicio de la actividad ***o*** y ***off(o)***, punto temporal que representa el fin de la actividad ***o***,

- $on(o) = t_i$, suponiendo que o se inicia en t_i

- *off(o)* $= t_j$, suponiendo que o se finaliza en t_j

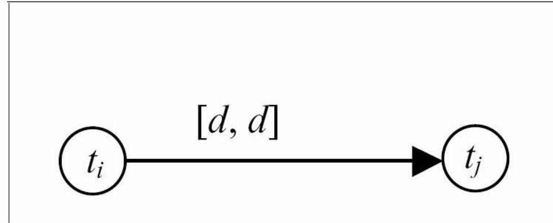

FIGURA 3.1: Representación gráfica de una actividad o con duración d.

3.3. Restricciones no disyuntivas

Son todas aquellas restricciones en las que interviene un único intervalo temporal.

Para especificar la **duración**, *dur*(o_i), de una operación usaremos la restricción temporal siguiente, *(on(o_i)* {*[d d]*} *off(o_i))*. Puesto que la restricciones se expresan mediante intervalos temporales, se pueden expresar restricciones en las que la duración de la actividad es fija, (por ejemplo una duración de 5, vendría dada por [5, 5]), o variable (por ejemplo si una operación puede durar entre 3 y 10 unidades de tiempo, el intervalo sería [3, 10]). Se asume, por simplicidad, que las duraciones van a ser fijas.

La relación de **precedencia** entre dos actividades o_i y o_j (denotada como $o_i \rightarrow o_j$), puede especificarse mediante las restricción temporal *(off(o_i)* {*[0 ∞]*} *on(o_j))*. Dicha restricción indica que desde que acabe la operación o_i hasta que comience la operación siguiente o_j debe transcurrir un periodo de tiempo ≥ 0 arbitrariamente grande.

Un trabajo J_i, i=1 $\ldots n$, estará formado por un conjunto de m operaciones $\{o_{ij}\}$, j=1 $\ldots m$, siendo n el número total de trabajos. Dichas operaciones estarán ordenadas mediante restricciones de precedecia entre ellas.

Así, dado un trabajo J_i, formado por m operaciones ordenadas $o_{i1}, o_{i2}, \ldots, o_{im}$, llamaremos ***firstJ***$_i$ a la primera operación del trabajo *Ji*, y ***last(J***$_i$***)*** a la última operación de dicho trabajo. Es decir:

- *first*$(J_i) = o_{i1}$
- *last*$(J_i) = o_{im}$

Para especificar el **tiempo de inicio** r *(readytime)*, el tiempo a partir del cual el trabajo J_i puede comenzar su ejecución, se usará la restricción *(**T0** {[r ∞]} on(first(J_i)))*, indicando que J_i puede comenzar a partir de r.

De la misma forma suponiendo un **tiempo límite** *dd (deadline)*, o tiempo máximo de finalización de J$_i$, estará especificado mediante la restricción ***T0*** *{[0 dd]} off(last(J_i))*, que significa que la última operación de J_i puede terminar como muy tarde en dd. En la Figura 3.2 aparece representado un trabajo J_i, formado por m operaciones, con un tiempo de inicio r y un *deadline dd*.

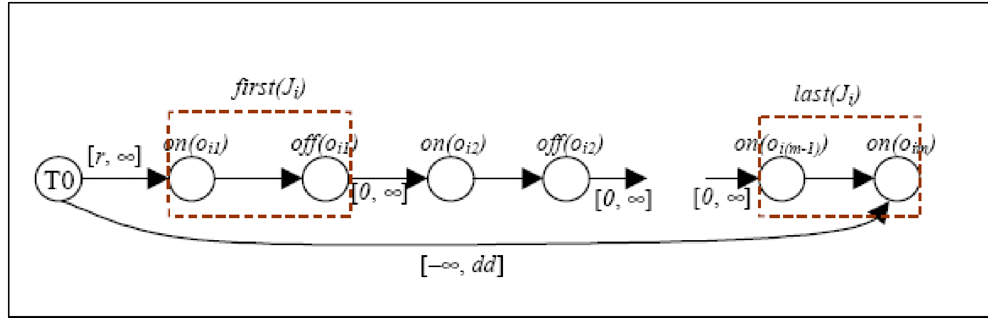

FIGURA 3.2: Restricciones temporales asociadas a un trabajo J_i.

Si la duración de las actividades es fija, entonces es posible simplificar la red de restricciones utilizando únicamente los puntos temporales que denotan los tiempos de comienzo de las operaciones, *on*(o_{ij}). De esta forma se reduce a la mitad el número de puntos temporales necesarios para representar todas las operaciones del *scheduling*. En la Figura 3.3 se puede ver la red de restricciones asociada a un trabajo J_i. En este caso, las relaciones de precedencia deben tener en cuenta la duración de las actividades correspondientes. Suponga dos operaciones o_{ij} y $o_{i(j+1)}$. Si se denota como d_{ij} a la duración de la operación o_{ij} y se asume que $o_{ij} \rightarrow o_{i(j+1)}$, entonces la restricción de precedencia entre ellas se puede especificar como *(on*(o_{ij}) $\{[d_{ij}\ \infty]\}$ *on*$(o_{i(j+1)})$*)*. También se debe tener en cuenta la duración de la última operación del trabajo (o_{im}), para especificar el *deadline* correspondiente. Si d_{im} es

duración de o_{im}, y dd el *deadline* asociado al trabajo J_i entonces la restricción temporal que expresa dicho *deadline* será *(**T0** {[0 dd+d_{im}]} off(last(J_i)))*.

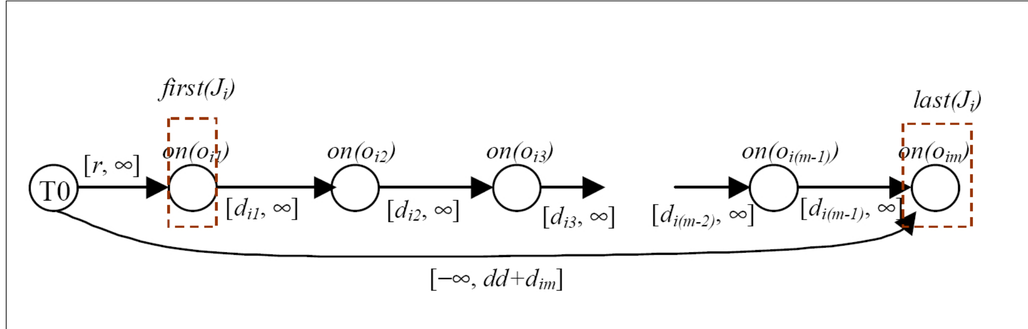

FIGURA 3.3: Representación de las operaciones mediante un único punto temporal.

3.3.1. Tiempos de preparación de las máquinas

Hay ocasiones en las que las máquinas requieren un cierto tiempo de preparación para ejecutar las operaciones, por ejemplo en el caso de que se trate de operaciones especiales, que requieran un cambio de configuración de recurso correspondiente (ajuste de parámetros o similar) para poder llevar a cabo dichas operaciones.

Este tipo de restricciones no se suelen considerar en problemas de *scheduling* puros, pero pueden representarse por medio de restricciones métricas sin disyunciones.

Los tiempos de preparación de las máquinas son conocidos a priori, puesto que conocemos las operaciones que van a ejecutarse sobre dichas máquinas. Por simplicidad, vamos a suponer unos tiempos de preparación fijos. En este caso, dado un recurso M_k, sobre el que se ejecuta la operación o_i, cuya duración es dur(o_i), y que requiere un tiempo de preparación *tp(o_i)*, simplemente añadiremos dicho tiempo a la duración de la actividad o_i, de forma que su nueva duración será: *dur(o_i) + tp(o_i)*.

Dado un trabajo J_i formado por la secuencia de m operaciones $o_{i1}, o_{i2}, \ldots, o_{im}$, con duraciones $d_{i1}, d_{i2}, \ldots, d_{im}$, que se ejecutan sobre las máquinas $M_1, M_2, \ldots, M_m$ respectivamente, cada una de las cuales necesita unos periodos de preparación $tp(o_{i1}), tp(o_{i2}), \ldots, tp(o_{im})$, las restricciones temporales asociadas serán:

- $(on(o_{ij})\ \{[(d_{ij} + tp(o_{ij}))\ \infty]\}\ on(o_{i(j+1)}))\ \forall\ J = 1 \ldots m\text{-}1$

En la Figura 3.4 aparece representado un trabajo formado por 4 operaciones $J = (o_1, o_2, o_3, o_4)$ que se ejecutan sobre las máquinas (M_1, M_2, M_3, M_4), que requieren unos tiempos de preparación de 2, 4, 3 y 8 respectivamente. Las duraciones de las actividades son 2, 6, 7 y 10.

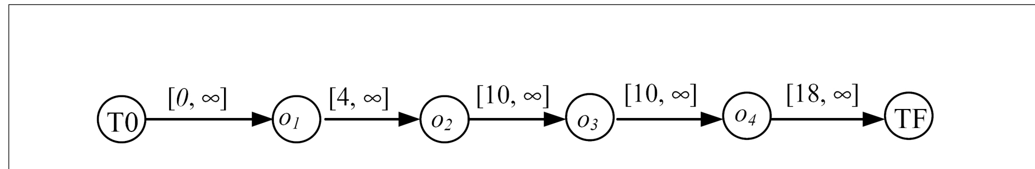

Figura 3.4: Grafo de restricciones para máquinas con tiempos de preparación.

3.4. Restricciones disyuntivas

Se deben al hecho de que cada recurso puede procesar una única operación cada vez, por lo tanto dadas dos operaciones, o_i y o_j, que se ejecutan sobre una misma máquina, o bien el orden es $o_i \rightarrow o_j$, o el inverso.

La restricción disyuntiva que refleja esta situación es $(on(o_i)\ \{[d_i\ \infty],\ [-\infty\ \text{-}d_j]\}\ on(o_j))$, siendo d_i y d_j las duraciones de las operaciones o_i y o_j respectivamente. El intervalo $[d_i\ \infty]$ expresa el orden de secuenciamiento $o_i \rightarrow o_j$, que significa que la operación o_i se realiza antes que la operación o_j, mientras que $[-\infty\ \text{-}d_j]$ indica el orden $o_j \rightarrow o_i$. La representación gráfica aparece en la Figura 3.5.

3.4.1. Patrón de flujo

Un patrón de flujo determina la secuencia de uso de las operaciones para cada máquina (ver 1.1). Puesto que cada máquina solamente puede procesar una operación cada vez, un patrón de flujo vendrá representado por un conjunto de restricciones disyuntivas para cada máquina.

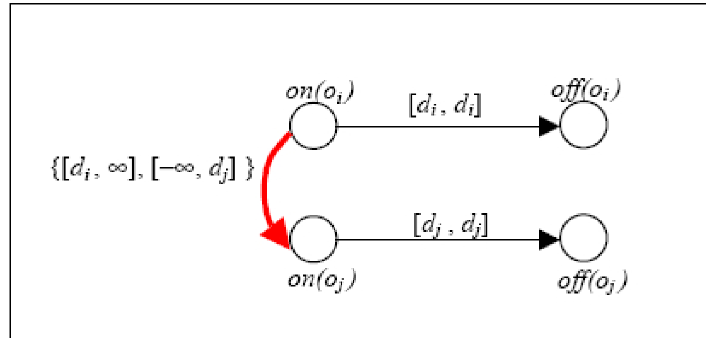

Figura 3.5: Restricción disyuntiva entre las operaciones o_i y o_j.

Los patrones de flujo más utilizados son los llamados *flow-shop* y *job-shop*. En la caso de ***flow-shop*** todos los trabajos utilizan los recursos en el mismo orden. Supongamos que tenemos 3 trabajos J_1, J_2, y J_3, formados por la secuencia de operaciones: $J_1 = (o_{11}, o_{12}, o_{13}, o_{14})$, $J_2 = (o_{21}, o_{22}, o_{23}, o_{24})$, $J_3 = (o_{31}, o_{32}, o_{33}, o_{34})$. Se dispone de 4 máquinas: m_1, m_2, m_3, m_4. Cada uno de los trabajos utilizan las máquinas en el orden (m_1, m_2, m_3, m_4). Las restricciones disyuntivas asociadas vienen representadas en la Figura 3.6, en donde el flujo para cada una de las máquinas está resaltado con un color distinto. Las duraciones de las actividades son fijas, por lo que pueden representarse mediante un único punto temporal. Los conjuntos D_i, son intervalos temporales que forman las restricciones entre pares de puntos, y que se representan sobre los arcos de la red de la Figura 3.6.

Si el patrón de flujo es ***flow-shop***, entonces cada trabajo puede tener una secuencia distinta de paso por las máquinas. En la Figura 3.7 aparecen representados 3 trabajos: $J_1 = (o_{11}, o_{12}, o_{13})$, $J_2 = (o_{21}, o_{22}, o_{23})$, $J_3 = (o_{31}, o_{32}, o_{33})$, y 3 máquinas m_1, m_2, m_3. La secuencia de uso de los trabajos es: para J_1: (m_1, m_2, m_3), para J_2: (m_2, m_1, m_3), y para J_3: (m_2, m_3, m_1). En la figura no se encuentran representadas todas las disyunciones, para una mayor claridad, pero están indicados los intervalos temporales que forman parte de las restricciones correspondientes para cada una de las máquinas.

Dados los siguientes elementos:

- n trabajos: $J_1, J_2, \ldots, J_n$,

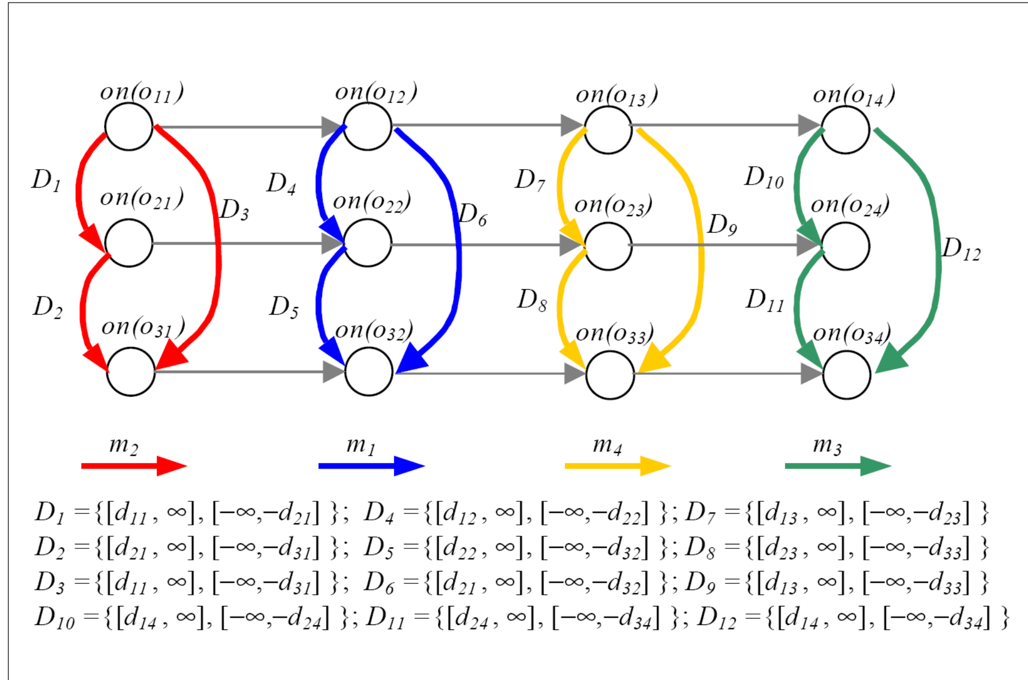

FIGURA 3.6: Representación de patrón de flujo para una instancia *flow-shop* 3×4.

- cada trabajo J_i está formado por la secuencia de m operaciones: $o_{i1}, o_{i2}, \ldots, o_{im}$, con duraciones $d_{i1}, d_{i2}, \ldots, d_{im}$ respectivamente,
- se dispone de m máquinas: $M_1, M_2, \ldots, M_m$ y cada trabajo J_i usa las máquinas según el orden $M_1, M_2, \ldots, M_m$,

las restriciones asociadas con el patrón de flujo *flow-shop* son las siguientes:

- *(on(o_{pj}) {[d_{pj} ∞], [$-\infty$ d_{qj}]} on(o_{qj}))* $\forall p, q = 1 \ldots n, p \neq q, \forall j = 1 \ldots m$

Dados los siguientes elementos:

- n trabajos: $J_1, J_2, \ldots, J_n$,
- cada trabajo J_i está formado por la secuencia de m operaciones: $o_{i1}, o_{i2}, \ldots, o_{im}$, con duraciones $d_{i1}, d_{i2}, \ldots, d_{im}$ respectivamente,

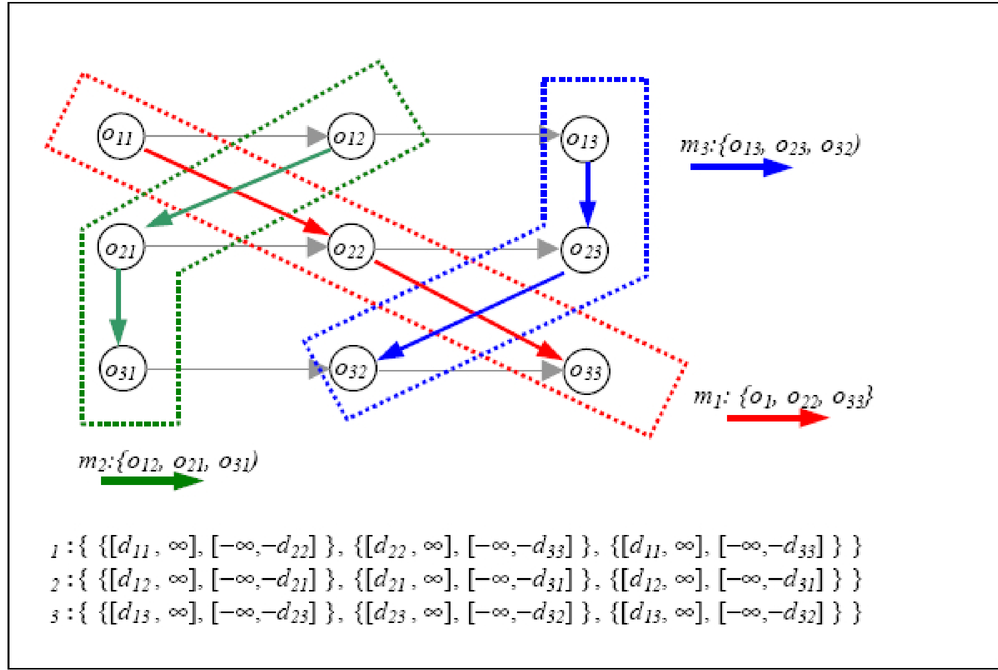

FIGURA 3.7: Representación de patrón de flujo para una instancia *job-shop* 3×3.

- se dispone de m máquinas: $M_1, M_2, \ldots, M_m$ y cada trabajo J_i usa las máquinas según un orden determinado $a_1, a_2, \ldots, a_m$. La secuencia $a_1, a_2, \ldots, a_m$ representa una permutación de las máquinas $M_1, M_2, \ldots, M_m$ y es distinta para cada trabajo. Por lo tanto, cada máquina M_k tiene asociado un conjunto de operaciones $\{o_{ij}\}$ tales que o_{ij} se ejecuta sobre M_k,

las restricciones temporales asociadas al patrón de flujo *job-shop* son las siguientes:

- *(on(o_{pj}) {[d_{pj} ∞], [−∞ d_{qj}]} on(o_{qj}))* con $p, q = 1 \ldots n, p \neq q, \forall o_{pj}, o_{qj} \in M_j$, *∀ j = 1 … m*

Nótese que el *flow-shop* es un caso particular del *job-shop*.

Capítulo 4

Implementación del modelo usando CCP

Para este trabajo de grado se construyeron dos aplicaciones que trabajan en conjunto. El motor de búsqueda de soluciones con la implementación del modelo expuesto en el capítulo 3, se hizo en el leguaje de programación *Mozart* versión 1.3.2. La interfaz gráfica para el usuario se implementó en *Java* versión 1.6, que se encarga de invocar al motor de búsqueda de soluciones realizado en *Mozart*. En este capítulo se presentan los detalles de la implementación del motor de búsqueda, y en el siguiente los de la interfaz.

4.1. Archivo de entrada

Para que el motor de búsqueda pueda resolver un problema, se le deben de pasar los datos del problema a través de un archivo de texto plano construido de la siguiente forma, como está sugerido en [5]:

- La primera línea debe contener el número de trabajos y el número de máquinas.
- Por cada trabajo hay una línea que lista el número de la máquina y el tiempo de procesamiento para cada operación del trabajo.

- Las máquinas son numeradas iniciando con 0.

4.2. Estructura de datos

- La estructura de datos utilizada para representar el problema es un registro.
- Este registro contiene un único campo llamado *schedule* que es una lista donde se almacenan todos los trabajos.
- En cada posición de la lista *schedule* hay un registro (con etiqueta *job*) que representa un trabajo.
- Cada trabajo contiene 3 campos: *rt*, *dt* y *oper*, que almacenan respectivamente el *ready time*, *deadline* y la lista de operaciones que componen al trabajo.
- En cada posición de la lista *oper* hay un registro (con etiqueta *o*) que representa una operación.
- Cada operación contiene 2 campos: *dur* y *maq*, que almacenan respectivamente la duración de la operación y la máquina en la que se va a realizar.

A continuación se presenta un ejemplo de una estructura de datos para un problema con 4 trabajos y 2 máquinas.

```
taskSpec(
   schedule:[job(dt:nil oper:[o(dur:60 maq:0) o(dur:30 maq:1)] rt:0)
             job(dt:nil oper:[o(dur:75 maq:1) o(dur:25 maq:0)] rt:0)
             job(dt:nil oper:[o(dur:15 maq:1) o(dur:10 maq:0)] rt:0)
             job(dt:nil oper:[o(dur:1 maq:0) o(dur:1 maq:1)] rt:0)])
```

4.3. Restricciones no-disyuntivas

En la implemtación de este trabajo de grado se impusieron 3 tipos de restricciones. Las restricciones de tiempo de inicio de los trabajos, las de tiempo de finalización y las de precendencia entre las operaciones de los trabajos.

Para imponer las restricciones no-disyuntivas sólamente se hizo uso de los operador `=<:` y `>=:` de *Mozart*, que sirven para crear restricciones de tipo $\leq$ y $\geq$ respectivamente entre dos variables de dominio finito A y B.

A continuación se presentan los algoritmos utilizados para imponer las restricciones no-disyuntivas:

Algoritmo 1 Restricciones de tiempo de inicio

1: **for all** $job \in Jobs$ **do**
2: $\quad oper \leftarrow job(1)$
3: $\quad readyTime \leftarrow job.rt$
4: $\quad oper.start \geq readyTime$
5: **end for**

Algoritmo 2 Restricciones de tiempo de finalizacion

1: **for all** $job \in Jobs$ **do**
2: $\quad oper \leftarrow job(last)$
3: $\quad deadLine \leftarrow job.dt$
4: $\quad oper.start \leq deadLine - oper.dur$
5: **end for**

Algoritmo 3 Restricciones de precedencia entre las operaciones

1: **for all** $job \in Jobs$ **do**
2: $\quad$ **for** $i \leftarrow 1, length(job) - 1$ **do**
3: $\quad\quad oper1 \leftarrow job(i)$
4: $\quad\quad oper2 \leftarrow job(i+1)$
5: $\quad\quad oper1.star + oper1.dur \leq oper2.start$
6: $\quad$ **end for**
7: **end for**

Finalmente también se impone la restricción `FD.assign min`, que asigna a todos los tiempos de inicio de las operaciones el mínimo valor posible.

4.4. Restricciones disyuntivas

Para imponer las restricciones disyuntivas se hizo uso de un propagador diseñado para ese propósito y que hace parte de los módulos del sistema de *Mozart*. Este propagador, que se puede invocar mediante la instrucción `Schedule.serialize` impone restricciones de tipo

```
(oper1.start + oper1.dur =<: oper2.start) +
(oper2.start + oper2.dur =<: oper1.start) =: 1
```

para todas las operaciones a ejecutarse en una misma máquina, en otras palabras, garantiza que cualesquiera dos operaciones que se ejecutan en la misma máquina no se solapan en el tiempo. Dicha instrucción, crea un propagador para cada máquina, evitando así, tener que crear un propagador para cada restricción entre dos operaciones (lo que generaría un número de propagadores cuadrático con respecto al número de operaciones procesadas por una máquina) [6].

4.5. Estrategias de propagación

Para mejorar el desempeño de la aplicación, se implementaron mecanismos de propagación propuestos en [7] para restricciones disyuntivas que redujeran los espacios de búsqueda. La idea es aprovechar las propiedades de las restricciones disyuntivas ya que se cuenta con la sisguiente información:

- Cuando el valor posible mas pequeño para *off(A)* excede el valor posible mas grande para *on(B)*, la operación A no puede preceder a la operación B, por lo tanto B debe preceder a A. A partir de esta propiedad se puede agregar una restricción. *off(B)* $\leq A$.
- Similarmente, cuando el mínimo de tiempo de finalización de B excede el máximo tiempo de inicio de A, B no puede preceder a A.
- Cuando ninguna de las dos actividades puede preceder a la otra, se detecta una contradicción.

4.6. Estrategias de distribución

[6] sugiere usar el serializador[1] `Schedule.firstsLastsDist`. Este serializador presenta varias cualidades. Entre ellas se encuentran:

[1]Un serializador es un distribuidor que serializa todas las tareas u operaciones en una máquina.

- Con este serializador, la profundidad un árbol de búsqueda crece linealmente con respecto al número de operaciones. Esto se debe, a que en cada nodo se ordenan varias operaciones a la vez, y no solamente dos, como es usual. Esto se hace restringiendo que una operación debe preceder todas las otras operaciones en una máquina.
- Presenta ventajas con respecto a estrategias de distribución de cuello de botella estáticas, ya que toma en cuenta los cambios de tamaño de los dominios en tiempo de ejecución para seleccionar el recurso que debe ser serializado.

4.7. Búsqueda de soluciones

El algoritmo de búsqueda implementado se encarga de buscar la solución óptima al problema que recibe como entrada. Para cumplir con este propósito se hace uso de la función `SearchBest` de *Mozart*, que se encarga de buscar una solución óptima de acuerdo a una función de costos que se le asigna [6].

`SearchBest` utiliza el esquema de ramificación y poda (*branch & bound*) para buscar la solución óptima y trabaja de la siguiente manera. Cuando se encuentra una solución, restringe todas las alternativas restantes del árbol de soluciones para que sea mejor que la solución encontrada, de acuerdo con el criterio establecido en la función de costos (en este caso el *makespan*). De esta manera se descartan las soluciones que sean peores a la que se tiene actualmente y se reduce el espacio de búsqueda.

Capítulo 5

Interfaz de la aplicación

La interfaz de la aplicación fue desarrollada en *Java* versión J2SE 1.6. Fue desarrollada en *Java* para aprovechar los beneficios que ofrece el lenguaje y los diferentes paquetes disponibles para hacer este tipo de desarrollos.

Para poder hacer los diagramas de Gantt que ilustren la solución encontrada, se hizo uso de la libreria *JFreeChart* versión 1.09. Esto facilitó y agilizó en gran medida el desarrollo de la interfaz. Adicionalmente, también se uso la libreria *LiquidLnF* para mejorar el aspecto visual de la aplicación.

La aplicación es realmente simple de usar, y actualmente el usuario solo puede realizar un acción, que corresponde a cargar un archivo que contiene las especificaciones de un problema de *scheduling* para que lo resuelva. Una vez ha resuelto el problema, despliega la solución en un diagrama de Gantt.

En las figuras 5.1 y 5.2 se muestra la interfaz de la aplicación. En la parte superior derecha de la figura 5.2 se puede ver el *makespan* de la solución encontrada, es el número que se encuentra donde aparece “Fin del scheduling”. En el caso de la figura 5.2, es 55.

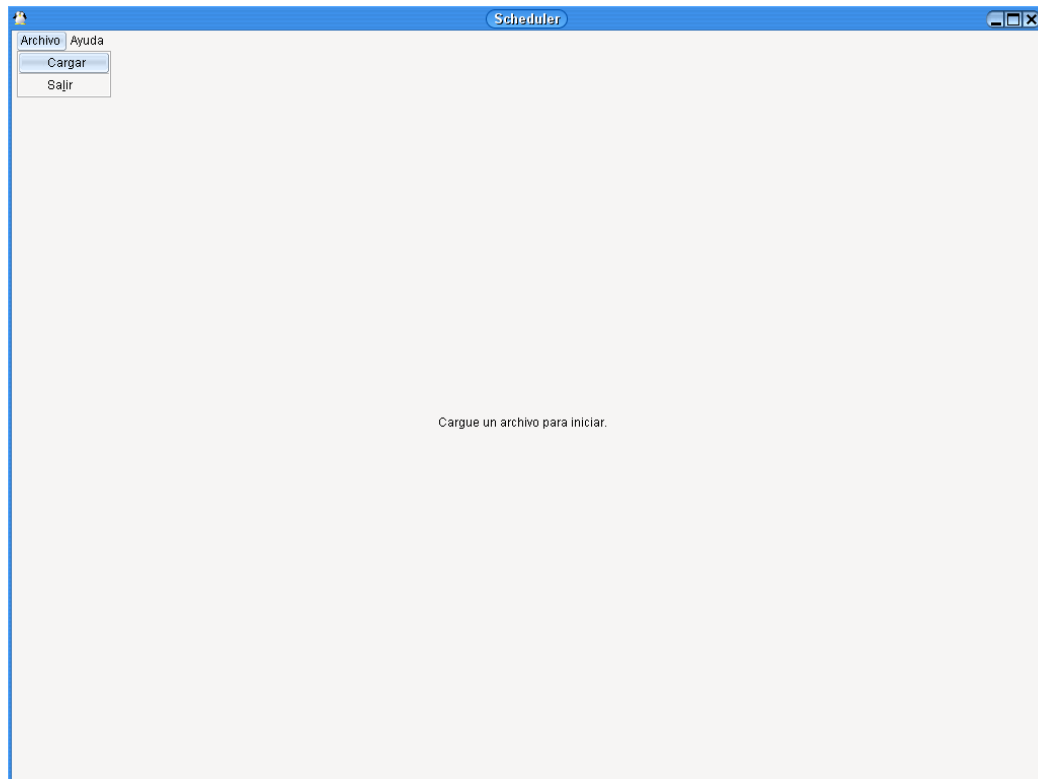

FIGURA 5.1: Vista de la aplicación cuando inicia.

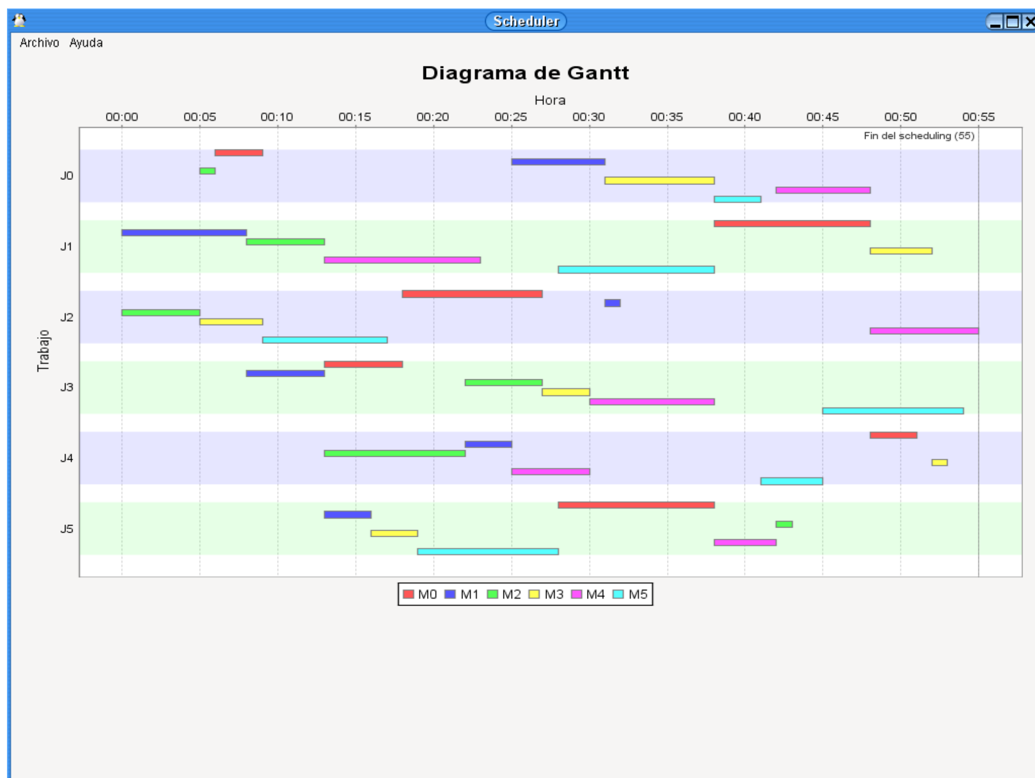

FIGURA 5.2: Vista de la aplicación con un diagrama de Gantt que representa la solución encontrada.

Capítulo 6

Experimentación y pruebas

Para probar la aplicación final, se tomaron problemas clásicos de *job-shop scheduling*, que se encuentran en la literatura. Los resultados se compararon contra resultados de otras implementaciones que abordan el problema desde diferentes paradigmas. El criterio de comparación es el tiempo, ya que el algoritmo implementado siempre arroja la solución óptima. Sin embargo, también se compara contra otros algoritmos que no alcanzan la solución óptima.

Los resultados fueron comparados con los de otros 5 algoritmos implementados utilizando distintos paradigmas. Los 5 algoritmos fueron probadas en arquitecturas de computadores similares a la usada para probar la implementación de este trabajo. Vale la pena aclarar que todos los algoritmos utilizan paradigmas estocásticos, en contraste con el paradigma utilizado en este trabajo de grado, que es determinístico. Esto ocasiona que los otros algoritmos, por ser estocásticos, no puedan garantizar la optimalidad de las soluciones encontradas. En la tabla 6.1 se presenta una lista de las implementaciones contra las cuales se comparan los resultados.

6.1. El problema *ft10*

Inicialmente las comparaciones con otras implementaciones se hicieron con respecto a un famoso problema de *job-shop scheduling* llamado *ft10* (también conocido como *mt10*). Este

Algoritmo	Paradigma
Yamada and Nakano [8]	Temple simulado
Gonçalves et˜al. [9]	Algoritmo genético
Croce et˜al. [10]	Algorítmo genético
Wang and Zheng [11]	Híbrido genético y temple simulado
Aiex et˜al. [12]	GRASP

Cuadro 6.1: Algoritmos con los cuales se compararon resultados y sus respectivos paradigmas de resolución

problema es conocido porque resulta especilamente difícil de resolver, de hecho, fue propuesto en 1963 por Fisher y Thompson [13], y no se consiguió encontrar la solución óptima hasta 1989 por Carlier y Pinson [14], a pesar de que cada nuevo algoritmo propuesto lo usaba como prueba [1]. El problema es de tamanño 10×10 (10 trabajos $\times$ 10 máquinas).

En la tabla 6.2 se puede ver los resultados obtenidos. La columna problema contiene el nombre del problema resuelto, la siguiente columna contiene la solución óptima, la columna tiempo contiene el tiempo en segundos que se demoró la implementación de este trabajo de grado en devolver la solución óptima. Las restantes columnas contienen los tiempos en segundos que se demoraron en arrojar resultados las otras implementaciones. Los tiempos presentados son el promedio de 5 iteraciones para resolver cada problema en una máquina con procesador AMD Turbo 3000+ de 1.59 GHz y 512 MB de RAM.

Problema	Solución optima	Tiempo (seg.)	Yamada	Gonçalves	Croce	Wang	Aiex
ft10	930	16.2	190	292	628[1]	44.37	4.05

Cuadro 6.2: Comparación de tiempos de respuesta para resolver el problema *ft10*.

Claramente se puede ver una superioridad en comparación contra la mayoría de los algoritmos. Solamente el algoritmo presentado en [12] tiene un mejor tiempo de respuesta.

[1]Este algoritmo no encontró la solución óptima, el *makespan* encontrado fue 946.

Además, el algoritmo implementado en este trabajo de grado cuenta con la ventaja de ser completo, y por tal motivo, puede asegurar que el resultado corresponde a la solución óptima.

6.2. Otros problemas

También se hicieron pruebas y comparaciones en base a otros problemas clásicos de la literatura. Los resultados se presentan el la tabla 6.3. Además se incluye la columna tamanño en donde se presenta el número de trabajos × número de máquinas que tiene cada problema.

Problema	Solución optima	Tamanño	Tiempo (seg.)	Gonçalves	Croce	Wang	Aiex
abz5	1234	10×10	11.5	–	–	–	2.53
abz6	943	10×10	2.4	–	–	–	2.26
la01	666	10×5	0.1	37	282	5.72	0.82
la06	926	15×5	0.1	99	473	12.29	8.0
la11	1222	20×5	1.2	197	717	25.94	3.14
la16	945	10×10	6.0	232	637[2]	29.43	2.27
la19	842	10×10	18.3	235	–	–	15.14
la20	902	10×10	50.3	235[1]	–	–	1.63

CUADRO 6.3: Comparación de tiempos de respuesta para resolver diversos problemas.

[1]Este algoritmo no encontró la solución óptima, el *makespan* encontrado fue 907.
[2]Este algoritmo no encontró la solución óptima, el *makespan* encontrado fue 979.

Capítulo 7

Conclusiones y trabajo futuro

7.1. Según los resultados obtenidos en el Capítulo 6 de Experimentación y pruebas

- Se puede ver de acuerdo a las pruebas realizadas, que el algoritmo implementado se comporta bien en comparación con otras implementaciones.
- Para problemas con un número pequeño de máquinas, siempre se pudo obtener la solución óptima.
- El algoritmo implementado tiene una ventaja con respecto a otros, y es que garantiza que la solución encontrada es la óptima.
- La dificultad de un problema, y por consecuencia el tiempo de respuesta para encontrar una solución, se ve enormemente influenciada por el número de máquinas distintas que procesan las operaciones. Mientras que el número de trabajos a procesar no tiene tanto peso en la dificultad para encontrar soluciones.

7.2. Conclusiones generales

- De acuerdo a los resultados obtenidos, se puede ver que el modelo en el cual se basa este trabajo de grado ha tenido un buen comportamiento con respecto a otras implementaciones, además resulta ser flexible, lo que permitiría realizar extensiones al modelo, para poder incluir otros aspectos propios de los problemas de *scheduling* del mundo real, tales como el tiempo de mantenimiento de las máquinas, la fabricación por lotes o los costos de producción.

- *Mozart* se presentó como un buen lenguaje para resolver problemas de *scheduling*, ya que la expresividad del lenguaje permite imponer restricciones de manera explícita e intuitiva. Además posee herramientas para trabajar sobre estos problemas, lo que facilita aún mas la implementación.

- Así como *Mozart* resultó ser una herramienta ideal para construir el motor de búsqueda de soluciones, también lo fue *Java* para construir una interfaz gráfica de usuario que pudiera presentar una interpretación gráfica de la solución, ya que cuenta con las herramientas necesarias para hacerlo, y que además reducen el tiempo de codifiación (gracias al paquete *JFreeChart*, que contiene una clase para generar diagramas de Gantt fácilmente).

7.3. Trabajo futuro

- Se pueden incluir en la implementación otros elementos que se encuentran en el modelo original planteado en [1], tales como el tiempo de descanso de las máquinas o la fabricación por lotes.

- Crear una plantilla en formato XML para facilitar el ingreso de los datos y que permita incluir mas variables del problema de *scheduling* que lo acerquen al mundo real. De esta forma se podrían construir problemas de *scheduling* complejos de manera intuitiva y explícita.

- Adicionar a la interfaz un asistente para crear nuevos archivos de entrada al motor de búsqueda de soluciones.

- Extender el modelo presentado en [1] para que un trabajo pueda tener varias operaciones que utilicen la misma máquina.
- Modificar la implementación hecha en *Mozart*, para que el algoritmo arroje la mejor solución encontrada en un tiempo determinado, que se le pasaría como parámetro.

Bibliografía

[1] María Isabel Alfonso. *Un modelo de integración de técnicas de clausura y CSP de restricciones temporales: Aplicación a problemas de Scheduling*. PhD thesis, Universidad de Alicante, 2001. URL http://www.dccia.ua.es/~eli/Tesis.pdf.

[2] K.R. Baker. *Introduction to sequencing and scheduling*. John Wiley & Sons Inc., 1974.

[3] M.S. Fox and N. Sadeh. Why is scheduling difficult? a csp perspective. In *Proceedings of the 9th European Conference on Artificial Intelligence, ECAI-90*, pages 754–767, 1990.

[4] Juan Francisco Díaz, Jesus A. Aranda, and James Ortiz. Consejero para la repartición de artículos. In *Proyecto crear*. Grupo AVISPA, 2004.

[5] J.E. Beasley. Or-library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

[6] Christian Schulte and Gert Smolka. *Finite Domain Constraint Programming in Oz. A tutorial*, 1.3.2 edition, June 2006.

[7] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. Incorporating efficient operations research algorithms in constraint-based scheduling. *Proceedings of the First International Joint Workshop on Artifficial Intelligence and Operations Research*, 1995.

[8] Takeshi Yamada and Ryohei Nakano. Job-shop scheduling by simulated annealing combined with deterministic local search. *Meta-heuristics: theory & applications*, page 237248, 1996.

[9] José Fernando Gonçalves, Jorge José de Magalhães Mendes, and Maurcio G.C. Resende. A hybrid genetic algorithm for the job shop scheduling problem. Technical report, AT&T Labs Research, 2002.

[10] Federico Della Croce, Roberto Tadei, and Giuseppe Volta. A genetic algorithm for the job shop problem. *Computers Ops Res.*, 22:15–24, 1994.

[11] Ling Wang and Da-Zhong Zheng. An e!ective hybrid optimization strategy for job-shop scheduling problems. *Computers & Operations Research*, 28:585–596, 2001.

[12] R.M. Aiex, S. Binato, and M.G.C. Resende. Parallel grasp with path-relinking for job shop scheduling. *Parallel Computing*, 29:393–430, 2003.

[13] J. F.Muth and G. L. Thompson. *Industrial Scheduling.* Prentice-Hall, 1963.

[14] J. Carlier and Pinson E. An algorithm for solving job shop problem. *Management science*, 35:164–176, 1989.