UNIVERSIDAD DE BUENOS AIRES

Facultad de Ciencias Exactas y Naturales

Departamento de Matemática

Tesis de Licenciatura

Un algoritmo para el Problema de Corte de Stock en Dos Dimensiones por Matching Iterado.

Ignacio Ojea

Director: Dra. Susana Puddu

2008

Agradecimientos

A la flia, por obvias y holgadas razones.

A Fabio y a Susana, por interesarme en la resolución de problemas aplicados, y por poner sus conocimientos en la materia a disposición de sus alumnos.

A Alejandro Rossetti y Aldo Pizzo, que me enseñaron el gusto por la matemática.

A mis compañeros: Manu, Nico, Cecilia, Javier, Fede, Julián y seguro que me olvido de un montón, por amenizar hasta las más largas jornadas. A Lucas, que con unos pocos comentarios dirimió en mi favor la batalla contra el C++.

Al resto de los amigos, que por suerte son unos cuantos, y no caben en este pequeño margen.

Índice

Resumen

El presente trabajo aborda el problema de optimización combinatoria conocido como *Corte de Stock en dos dimensiones*[1], mediante el estudio del algoritmo presentado por Fritsch y Vornberger en [2]. Este problema se encuentra con frecuencia en diversas áreas de la industria: del vidrio, del papel, metalúrgica, etc. Fritsch y Vornberger, por ejemplo, diseñaron su algoritmo para la industria del vidrio. En nuestro caso, el objetivo final es desarrollar una implementación satisfactoria para ser utilizada en una carpintería.

En la primera parte del trabajo presentamos la teoría de los problemas $\mathcal{NP}$, $\mathcal{NP} - Completos$ y $\mathcal{NP} - Hard$, con el objeto de encuadrar el problema en ella. Para esto, introducimos los conceptos de Máquina de Turing Determinísitica y No-Determinística, de reducción polinomial y de reducción Turing. Finalmente, demostramos el Teorema de Cook y, a partir de él, una serie de resultados que llevan a concluir que el problema aquí estudiado pertenece a la clase de los NP-Hard.

La segunda parte está dedicada al estudio del algoritmo de Fritsch y Vornberger, sus virtudes y sus limitaciones. En la sección 4 presentamos las herramientas fundamentales para el desarrollo del método y exponemos sus lineamientos generales, tal como lo describen sus autores. La sección 5 está dedicada al comentario de algunos aspectos interesantes de la implementación. Por otra parte, dado que el método en su versión original resultó ser poco eficaz para el caso que nos interesa, exponemos las variantes que introdujimos para mejorar su rendimiento. Finalmente, en la sección 6 comentamos algunos resultados obtenidos.

[1] En la bibliografía: *Two-dimensional cutting stock problem.*

1. Preliminares

1.1. El Problema

CORTE DE STOCK (CSP)

Dada una demanda de piezas rectangulares $\{r_i\}$ $i = 1, ..., n$, no necesariamente orientadas, y una sucesión idealmente infinita de rectángulos grandes de stock R (todos iguales), se quiere diseñar un patrón para cortar las piezas de demanda de los rectángulos de stock, de manera de utilizar la menor cantidad posible de estos últimos. Puesto que todas las piezas son rectangulares, para definirlas bastará con dar sus dimensiones. De este modo, $r_i = (a_i, l_i)$, $i = 1, ..., n$ y $R = (A, L)$, donde las primeras coordenadas son los anchos y las segundas los largos. Todos los cortes deben ser de tipo *guillotina*.

La orientación de las piezas dependerá de las características del material. En el caso del vidrio nada obliga a fijar una orientación a priori. De este modo, la pieza r_i puede ser tomada, indistintamente, como el par (a_i, l_i) o como el par (l_i, a_i). En la carpintería, los rectángulos de stock son planchas de aglomerado enchapadas en melamina. Este enchapado puede ser imitación madera o de color liso. El caso liso es análogo al vidrio. Cuando la melamina es imitación madera, en cambio, las piezas de demanda deben respetar la *veta* y tienen, por lo tanto, ancho y largo fijos, sin admitir rotaciones.

La restricción más importante que debe ser tenida en cuenta para el diseño del patrón es la de que sólo son admisibles los cortes de tipo *guillotina*, es decir aquellos que cortan un rectángulo en dos, de lado a lado (Figura 1) y sólo en sentido vertical u horizontal, pero no oblicuo. Esta condición es cierta tanto para la industria del vidrio como para la carpintería, así como también para algunas ramas de la industria metalúrgica, y viene determinada por las características de las máquinas que realizan el corte. Si bien esto puede ser visto como una fuerte limitación, dota al problema de una estructura que puede, y debe, ser explotada para el desarrollo de algoritmos eficientes.

Una *solución* al problema será una secuencia de cortes de guillotina, como los de la figura 1 que den como resultado las piezas del pedido.

Fritsch y Vornberger debieron tomar en cuenta otro condicionamiento, dado por la fragilidad del material: las planchas de stock de vidrio son mucho más largas que anchas. Esto obliga a que los primeros cortes se realicen paralelos al lado de menor longitud, puesto que un corte demasiado largo puede romper la pieza con facilidad. Esta restricción no se encuentra en el corte de maderas. Veremos, sin embargo, que las técnicas utilizadas para atender este inconveniente ayudan, de manera natural, a mejorar el rendimiento de las planchas de aglomerado.

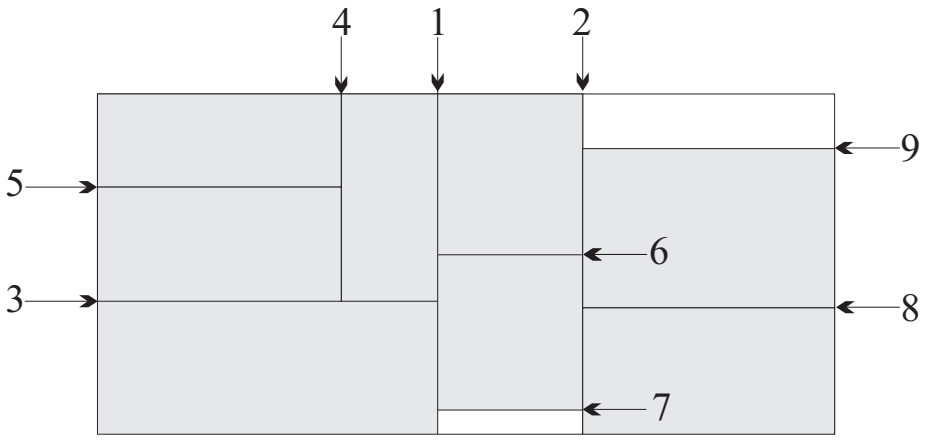


Figura 1: Todos los cortes deben realizarse de lado a lado.

1.2. Optimización Combinatoria

Un problema de optimización combinatoria puede plantearse de la siguiente manera: dado un conjunto finito $\mathcal{S}$ y una función $f : \mathcal{S} \to \mathbb{R}$ se desea encontrar el valor mínimo de f. La existencia de este mínimo (absoluto) es consecuencia directa de la finitud de $\mathcal{S}$. En la mayor parte de los casos interesará conocer, específicamente, el elemento $s \in \mathcal{S}$ en el cual se realiza el mínimo. Aclarado esto, aceptamos que una formulación general de un problema Π de optimización combinatoria es:

$$\Pi : \left\{ \begin{array}{c} \min f(s) \\ s \in \mathcal{S} \end{array} \right. \tag{1}$$

Llamaremos a $\mathcal{S}$ el conjunto de *soluciones factibles*, o *conjunto de factibilidad* de Π.

Este planteo abarca gran cantidad de problemas, de muy diversa naturaleza. El modo de encarar su resolución dependerá de las características propias de cada problema, definidas por la estructura y el contenido del conjunto $\mathcal{S}$ y de la función f.

Para aclarar algunos conceptos, presentamos uno de los ejemplos clásicos de la teoría: el conocido como Problema del Viajante:

`TSP`[2]

> Se tienen n ciudades, todas interconectadas entre sí, y se conoce la matriz de distancias $(d_{ij})_{1 \leq i,j \leq n}$, donde d_{ij} es la distancia de la ciudad i a la ciudad j. Se quiere planificar un viaje que permita visitar todas las ciudades, pasando una sola vez por cada una de ellas, y regresando finalmente a la ciudad de partida, de manera tal que la distancia total recorrida sea mínima. Más formalmente, buscamos un circuito que recorra las ciudades, que en el paso i

[2]Por sus siglas en inglés: Traveling Salesman Problem

de nuestro viaje pase por la ciudad $\alpha(i)$, de manera que la siguiente expresión sea mínima:

$$\sum_{i=1}^{n-1} d_{\alpha(i)\alpha(i+1)} + d_{\alpha(n-1)\alpha(1)} \tag{2}$$

En este caso, el conjunto $\mathcal{S}$ está formado por todos los posibles circuitos que recorren las n ciudades retornando a la original habiendo pasado una vez por cada una, que no es otra cosa que el grupo de permutaciones de n elementos S_n, mientras que la función f aplicada a un circuito dado es simplemente la suma de las distancias de los caminos utilizados en el circuito: ni más ni menos que la expresión dada en 2.

En el caso del `CSP`, el conjunto $\mathcal{S}$ está dado por todos los posibles patrones de corte que permiten extraer de las rectángulos de stock la totalidad de las piezas demandadas.

En principio, el hecho de que el conjunto $\mathcal{S}$ sea finito podría llevar a suponer que la resolución de un problema de este tipo no representa ninguna dificultad: basta evaluar la función f en cada elemento de $\mathcal{S}$ y elegir el mínimo. Sin embargo, como puede apreciarse en los ejemplos anteriores, la cantidad de elementos de $\mathcal{S}$, aunque finita, puede llegar a ser enormemente grande. En el `TSP`, un sencillo argumento combinatorio basta para verificar que, para n ciudades, la cantidad de circuitos posibles es $n!$. De este modo, si hubiese 50 ciudades para visitar, el número de circuitos sería del orden de 10^{64}. En el caso del problema de Corte de Stock, la situación es similar. La verdadera dificultad de los problemas de optimización combinatoria es, justamente, que el número de elementos del conjunto $\mathcal{S}$ crece exponencialmente con el *tamaño* del problema. Un análisis exhaustivo de dichos elementos resulta, por lo tanto, impracticable, pues requeriría de años (si no siglos) de procesamiento en una computadora moderna. Se hace necesario, entonces, desarrollar algoritmos particulares que aprovechen la estructura propia de cada problema para alcanzar una solución.

Importantes avances se han realizado en este sentido en los últimos cincuenta años. Muchos problemas han sido resueltos mediante algoritmos eficientes. Otros, sin embargo, como el `TSP` o el problema de Corte de Stock, siguen resultando *intratables*: pese a haber sido abordados desde diferentes enfoques no se ha logrado desarrollar algoritmos polinomiales que encuentren soluciones óptimas. Esta resistencia a ser resueltos ha hecho que los especialistas en el tema adquiriesen la casi certeza de que la dificultad, la *intratabilidad*, es intrínseca de ciertos problemas, y no sólo un espejismo ocasionado por la incapacidad para resolverlos. Esta convicción ha motivado un gran desarrollo teórico y ha desembocado en una de las conjeturas más importantes de la matemática moderna.

En lo que sigue, definiremos algunos conceptos que nos permitirán presentar una breve introducción a esta teoría.

1.3. Algoritmos y Complejidad

Hasta aquí hemos hablado del *tamaño* de un problema y de la *eficiencia* de un algoritmo sin precisar a qué nos referimos con estas palabras. Para formalizar estos conceptos, empecemos por revisar nuestra definición de problema.

Un problema general de optimización combinatoria, como lo estudiamos hasta ahora, viene dado por un conjunto $\mathcal{S}$ y un a función f, definida sobre $\mathcal{S}$. Ahora bien, lo que distinguirá a un problema de otro será, principalmente, la naturaleza de los elementos de $\mathcal{S}$. Éstos elementos vendrán dados por una descripción general de sus características. Tomando en cuenta este hecho, damos la siguiente:

Definición 1.1 *Un* **problema** *será una expresión que:* (1) *presente una lista de parámetros, detallando sus caracteristicas;* (2) *enuncie una propiedad que deban cumplir dichos parámetros. Llamaremos* **instancia** *del problema a una especificación de los valores de los parámetros. Dada una instancia particular, el problema en sí consistirá en decidir si cumple o no con la propiedad dada por* (2).

Por ejemplo: en el caso del `TSP`, los parámetros que definen el problema son, por un lado, el conjunto de las ciudades $C = \{1, 2, ..., n\}$ y, por el otro, la matriz de distancias $(d_{ij})_{1 \leq i,j \leq n}$. Una solución factible será un permutación α de las ciudades, que definirá un circuito $\{\alpha(1), \alpha(2), ..., \alpha(n)\}$. Finalmente la propiedad que se quiere verificar es que la suma de las distancias dada en (2) sea mínima. Una *instancia* del `TSP` es, por ejemplo, la dada por el conjunto $C = \{1, 2, 3, 4\}$ y las distancias $d_{12} = 3$, $d_{13} = 5$, $d_{14} = 13$, $d_{23} = 1$, $d_{24} = 5$, $d_{34} = 3$.

Para el problema de Corte de Stock, los parámetros son los pares (A, L) y (a_i, l_i) con $1 \leq i \leq n$, que dan el ancho y el largo de los rectángulos de stock R y de las piezas de demanda, respectivamente, y una variable binaria O que indicará si las piezas pueden o no ser rotadas. Una solución factible del problema vendrá dada por una distribución de las piezas del pedido que pueda ser realizada mediante cortes de guillotina y respete las dimensiones de R. La propiedad que debe verificarse una solución óptima es que la cantidad de rectángulos de stock R utilizados sea mínima.

Obsérvese que esta definición de *problema* es más general que la anterior, pues se extiende también a ejemplos que no pertenecen al dominio de la optimización combinatoria. E.g.:

`PRIMO`

Dado un entero positivo n, se quiere decidir si es primo o compuesto.

Este problema se acomoda perfectamente a nuestra última definición, siendo el único parámetro el número n y la propiedad acerca de la que se pregunta la de ser primo. Sin

embargo, el problema no es susceptible, al menos de manera natural, de ser planteado como la minimización de una función sobre un conjunto finito.

Un ***algoritmo*** será una secuencia de instrucciones para resolver un problema. Concretamente: un programa de computadora. Diremos que un algoritmo *resuelve* un cierto problema Π si tenemos garantizado que, al serle ingresada (como input) una instancia particular de Π, dará como resultado, en un tiempo finito, una solución para la instancia dada.

El ***tamaño*** de una instancia vendrá dado por la cantidad de datos que son necesarios para definirla. Especificamente, podríamos medir el tamaño de una instancia en una computadora por la cantidad de bytes que hacen falta para almacenarla en la memoria. En general diremos, por ejemplo, que una instancia del `TSP` es más grande que otra si involucra más ciudades. Análogamente, en el `CSP`, cuantas más piezas tenga un pedido, mayor será la instancia que definan.

Dado un algoritmo que resuelve un cierto problema Π, está claro que el tiempo requerido para alcanzar la solución variará de una instancia a otra. En particular, cuanto mayor sea el *tamaño* de una instancia, más tiempo necesitará el algoritmo para encontrar la solución correspondiente. Lo que nos interesará conocer, entonces, para decidir si un algoritmo es más o menos eficiente que otro, es *cuánto* más tardará en encontrar la solución para un cierto incremento en el tamaño de la instancia. Es decir: cómo es la relación entre el tamaño del input y el tiempo que tarda el algoritmo en dar una solución. Ahora bien: este *tiempo*, estrictamente hablando, no será intrínseco del algoritmo, sino que dependerá también de las características técnicas de la computadora en que se lo utilice. Para salvar esta ambigüedad tenemos el concepto de *complejidad*.

La ***complejidad*** de un algoritmo estará dada por el número de operaciones que realiza (sumas, productos, asignaciones de variables, comparaciones, etc.), en función del tamaño del input. El tiempo efectivo que necesite un algortimo para resolver una cierta instancia de un problema en una computadora dada dependerá, entonces, de su complejidad y del tiempo que tarde dicha computadora en realizar cada operación.

En general, diremos que una función $f(n)$ es *del orden de* otra función $g(n)$, y lo notaremos $f(n) = \mathcal{O}(g(n))$, si existe una constante positiva C tal que $|f(n)| \leq C|g(n)|$ $\forall n \geq 0$. De este modo, diremos que un algoritmo es ***polinomial*** si existe un polinomio p tal que su función de complejidad es un $\mathcal{O}(p(n))$, donde n es el tamaño del input. Por otro lado, y aún cuando existen complejidades intermedias, que no responden a un comportamiento polinomial ni exponencial, diremos que un algoritmo es ***exponencial*** si su complejidad no puede ser acotada por un polinomio. Típicamente, los algoritmos que examinan todas las posibles soluciones de un problema (todos los elemenentos de $\mathcal{S}$) son exponenciales. Finalmente, consideraremos que un algoritmo es *eficiente* si es polinomial y que es *ineficiente* si es exponencial. Cabe destacar que, para problemas de tamaño n chico, un algoritmo con complejidad, digamos, n^5 puede ser más lento que otro con complejidad del orden de 2^n (simplemente porque $n^5 < 2^n$ para n chico) [3].

[3] Una interesante tabla comparando complejidades puede consultarse en [3], p.7

Sin embargo, dado que lo que nos interesa estudiar es el comportamiento del algoritmo para tamaños grandes, donde el tiempo requerido para hallar la solución puede resultar inaceptable en la práctica, podemos considerar que nuestra definición de *eficiente* es razonable.

Los problemas considerados *intratables* son aquellos para los cuales no se conocen algoritmos eficientes, como el `TSP`, el `CSP` y algunos otros que describiremos más adelante. La teoría de los problemas $\mathcal{NP}$ y $\mathcal{NP} - Completos$, brinda herramientas que permiten *comparar* problemas, y arriba a algunos resultados que, si bien no son definitorios, inducen a sospechar que la *intratabilidad* es intrínseca a ciertos problemas y, por lo tanto, independiente del enfoque y de las técnicas que puedan utilizarse para su resolución.

2. Problemas $\mathcal{NP}$ y $\mathcal{NP} - Completos$

La teoría de la NP-Completitud, iniciada por Cook, estudia la intratabilidad de los problemas descriptos por la definición 1.1. La herramienta fundamental de la teoría será el concepto de *reducción polinomial*, que permitirá comparar problemas permitiendo llegar a resultados del tipo: el problema Π_1 es al menos tan difícil como el problema Π_2. La noción de *dificultad* remite directamente a la de complejidad. Consideraremos que Π_1 es tanto o más difícil que Π_2 cuando la existencia de un algoritmo eficiente para resolver Π_1 garantice la existencia de otro algorimo, también eficiente, para Π_2.

Para formalizar la noción de algoritmo definiremos la Máquina de Turing Determinística, que asociaremos a la clase $\mathcal{P}$, de los problemas ya resueltos por algoritmos polinomiales. Análogamente, el estudio de la Máquina de Turing No Determinística dará lugar a la definición del conjunto de problemas $\mathcal{NP}$. Un problema será $\mathcal{NP} - Completo$ (lo que consideramos *intratable*), cuando sea al menos tan difícil como cualquier otro problema $\mathcal{NP}$.

Por una cuestión de conveniencia, la teoría se centra en los llamados *problemas de decisión*. Esto es: aquellos que contemplan sólo dos posibles soluciones, dadas por las repuestas *sí* y *no*.

2.1. Problemas de Decisión

Definición 2.1 *Remitiéndonos a nuestra definición general dada en 1.1, diremos que un problema* Π *es de* **decisión** *si el conjunto de todas sus posibles instancias,* $\mathbf{D}_\Pi$*, contiene un sunconjunto distinguido* $\mathbf{Y}_\Pi$*, de* instancias positivas*, de manera tal que el problema consiste en, dada una instancia* I *de* Π*, decidir si pertenece o no a* $\mathbf{Y}_\Pi$.

Un ejemplo clásico de problema de decisión es el problema `PRIMO`, definido en 1.3.

Observemos que todo problema de optimización combinatoria dado por un par (S, f), admite naturalmente una *versión de decisión*, que consiste en, fijada una constante $B \in \mathbb{R}$, decidir si existe o no un elemenento $s \in S$ tal que $f(s) \leq B$. Como puede apreciarse, la versión de decisión no es más que una formulación relajada del problema original.

De este modo, una instancia del *Problema del Viajante de Decisión* (`TSPD`) estará dada por el conjunto de ciudades $C = \{1, 2, ..., n\}$, la matriz de distancias $(d_{ij})_{1 \leq i,j \leq n}$ y la cota $B \in \mathbb{R}$, y se preguntará si existe una cierta permutación α de las ciudades de manera que:

$$\sum_{i=1}^{n-1} d_{\alpha(i)\alpha(i+1)} + d_{\alpha(n-1)\alpha(1)} \leq B$$

Para el caso del problema de Corte de Stock, el planteo de la versión de decisión es

completamente análogo: dado un número $B \in \mathbb{N}$ se pregunta si existe un patrón de corte que disponga las piezas en a lo sumo B placas.

Es importante remarcar que no todos los problemas de decisión son formulaciones relajadas de problemas de optimización combinatoria. En cuanto a estos últimos, la observación funtamental que justifica el desarrollo de la teoría es que, siendo la función f fácil de evaluar, la versión de decisión no será nunca *más difícil* que la versión de optimización: Si tuviésemos un algoritmo que encontrase el $s \in S$ en que f alcanza su mínimo valor, bastaría con evaluar f en s y comparar el resultado con la cota B para resolver el problema de decisión. Es decir: la versión original de un problema de optimización combinatoria es siempre al menos tan difícil como su versión de decisión.

Un ejemplo clásico, y como veremos más adelante fundamental, de problema de decisión que no proviene de un problema de optimización es el siguiente:

SAT[4]

Dado un conjunto de variables booleanas (que pueden tomar valor `VERDADERO` o `FALSO`), $X = \{x_1, x_2, ..., x_n\}$, llamameros *literal* a cualquiera de las expresiones x_i ó $\bar{x_i}$, para $i = 1, ..., n$, donde $\bar{x_i}$ es la negación de x_i. Sobre el conjunto de los literales definimos dos operaciones, dadas por los operadores lógicos de disyunción (ó inclusivo), que notaremos aditivamente ($+, \sum$), y de conjunción (y), que notaremos multiplicativamente ($\cdot, \prod$). Una *cláusula* será una expresión que vincule a algunos de los literales de X mediante disyunciones. Una *sentencia* sobre X será un conjunto de cláusulas vinculadas por conjunción.Dada una expresión ξ sobre un conjunto X, el problema `SAT` consiste en decidir si existe una asignación de valores para las variables de X de manera que ξ resulte verdadera.

Así, las expresiones:

$$(x_1 + \bar{x_2} + x_3) \cdot (x_2 + x_4) \cdot (\bar{x_1} + \bar{x_3} + \bar{x_4}) \tag{3}$$

y

$$(\bar{x_1}) \cdot (x_1 + x_2) \cdot (\bar{x_2} + x_3) \cdot (\bar{x_3} + x_4) \cdot (x_1 + \bar{x_4}) \tag{4}$$

serán sentencias sobre el conjunto de variables $\{x_1, x_2, x_3, x_4\}$.

El ejemplo 3 admite una tal asignación, poniendo: $x_1 = x_2 = x_4 =$ `V` y $x_3 =$ `F`. Contrariamente, la expresión del ejemplo 4 es imposible de satisfacer. Para verificarlo, intentemos fabricar una asignación que haga verdadera a la expresión: la primera cláusula nos fuerza a definir $x_1 =$ `F`. Y esto, a su vez, nos obliga a definir sucesivamente $x_2 =$ `V`, $x_3 =$ `V` y $x_4 =$ `V`. Pero bajo esta asignación, la última cláusula resulta falsa, y por lo tanto también resulta falsa la expresión.

[4]Del inglés: *satisfiability*, que podría traducirse como: susceptible de ser satisfecho.

2.2. Lenguajes y esquemas de codificación

Para poder formalizar el concepto de algoritmo, necesitaremos primero precisar la noción de problema en términos de su ingreso en una computadora.

Dado un conjunto de símbolos Ω, que llamaremos *alfabeto*, denotaremos por Ω^* al conjunto de todas las posibles cadenas (palabras) que puedan formarse con los elementos de Ω. En una computadora, los símbolos admitidos son sólo el 0 y el 1. Las posibles *palabras* serán entonces todas las cadenas de ceros y unos, incluyendo la cadena vacía. Estas cadenas pueden representar números expresados en binario, o caractéres, según alguna tabla de codificación como la ASCII.

Un ***lenguaje*** sobre Ω será cualquier subconjunto $\mathcal{L}$ de Ω^*. Observemos que cualquier información o programa que se cargue en una computadora es un lenguaje en $\{0,1\}^*$. Un *esquema de codificación*, será una serie de reglas para indicar cómo deben interpretarse los diferentes tipos de datos y cómo deben seguirse las instrucciones del programa. En particular, una vez fijado un alfabeto Ω y un esquema de codificación $\mathbf{e}$, un determinado problema Π estará dado por todas las *palabras* en Ω^* que representen alguna instancia de Π. Llamaremos *lenguaje asociado al problema* Π, al conjunto de las representaciones de las instancias positivas de Π.

$$\mathcal{L}(\Pi, \mathbf{e}) = \{x \in \Omega^* : x \text{ codifica según } \mathbf{e} \text{ a una instancia } I \in \mathbf{Y}_\Pi\} \tag{5}$$

Puesto que la función de un esquema de codificación es especificar la forma en que deben interpretarse los datos y las instrucciones (tengamos en cuenta que en el idioma básico de una computadora toda la información está dada por cadenas de unos y ceros), el *tamaño* de una instancia, es decir el espacio que ocuparán los valores de los parámetros que la definen, dependerá del esquema de codificación que se utilice. Podemos asumir, sin embargo, que bajo cualquier esquema *razonable* el *tamaño* de una instancia se mantendrá controlado. Más específicamente, si según los criterios de un esquema $\mathbf{e_1}$, una instancia I ocupa n bytes, bajo las especificaciones de otro esquema $\mathbf{e_2}$, I no debería ocupar *mucho más* que n bytes. Esto último puede formalizarse de la siguiente manera: si el tamaño de I bajo $\mathbf{e_1}$ es n, entonces existirá un polinomio p de manera que el tamaño de I bajo cualquier otro esquema *razonable* $\mathbf{e_2}$ sea menor o igual que $p(n)$. El concepto de *esquema razonable* es algo huidizo. De hecho, no conocemos una definición precisa. Sin embargo, convendremos en que un *esquema razonable* es aquel que, por un lado se limita a retener la información escencial de una instancia, sin agregar datos superfluos y, por el otro, lo hace de manera que posibilite la decodificación de la información almacenada, es decir: que para cada componente de una instancia admite un algoritmo polinomial que la interpreta. Puede verse un ejemplo detallado en [3], página 21.

Si admitimos esto último, la eficiencia de una algoritmo, en términos de *polinomialidad* o *exponencialidad*, será independiente del esquema utilizado. De este modo, el *lenguaje asociado* a un problema Π puede pensarse como dependiendo exclusivamente del problema, y no del esquema. Cada problema quedará entonces asociado directamente

a un lenguaje:

$$\mathcal{L}(\Pi, \mathbf{e}) = \mathcal{L}(\Pi) \tag{6}$$

2.3. Máquina de Turing Determinística y problemas $\mathcal{P}$

El modelo que utilizamos para formalizar la noción de computadora es el de una Máquina de Turing Determinística (`MTD`). Consta de tres partes, esquematizadas en la Figura 2:

1. Un Controlador de Estados, que admite cualquier cantidad finita de estados.
2. Un Lecto-Grabador.
3. Una Cinta divida en casilleros numerados, idealmente infinita.

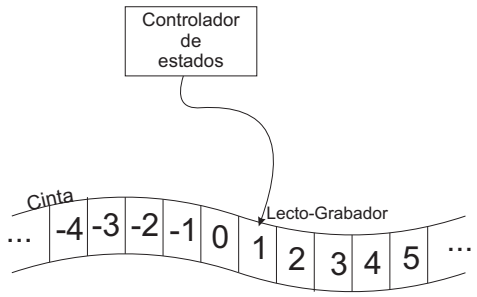


Figura 2: Máquina de Turing Determinística

Un *programa*[5] para una Máquina de Turing Determinísitica, es decir: un algoritmo dentro de nuestro modelo de computadora, vendrá dado por:

1. Un conjunto finito de símbolos Γ, dentro del cual distinguiremos un subconjunto Ω con los símbolos destinados a describir el input y un elemento especial b en $\Gamma \setminus \Omega$ para el *espacio en blanco*.
2. Un conjunto finito Q de posibles estados, conteniendo tres estados distinguidos: el estado inicial q_0 y los estados finales q_Y (respuesta afirmativa) y q_N (negativa).
3. Una función de transición $\delta : (Q \setminus \{q_Y, q_N\}) \times \Gamma \to Q \times \Gamma \times \{-1, 1\}$. Esta función representa las distintas instrucciones del programa. La asignación $\delta(q, \alpha) = (q', \beta, \Delta)$ indica que si el Controlador de Estados se encuentra en q y el Lecto-Grabador apuntando a un casillero con el símbolo α, entonces el programa deberá: borrar α para escribir en su lugar el símbolo β, moverse sobre la cinta un casillero hacia la derecha o hacia la izquierda según sea $\Delta = 1$ o $\Delta = -1$, y finalmente ubicar el Controlador de Estados en q'. Cuando el Controlador de Estados se encuentra en q_Y o en q_N el programa termina, dando como respuesta: *SI* y *NO* respectivamente.

[5] Pueden verse algunos ejemplos de programas definidos sobre Máquinas de Turing en [7] y en [11]

Para nosotros el input de un programa será una instancia de un problema de dicisión. De aquí que los únicos dos estados terminales de Q sean q_Y y q_N. Inicialmente, la cinta tendrá el símbolo blanco b en todos sus casilleros. El primer paso de todo programa será cargar la cadena de símbolos que representen a la instancia dada en los primeros lugares positivos de la cinta. Luego el Lecto-Grabador se ubicará en el casillero 1 y el Controlador de Estados en q_0. A partir de allí se comportorá según las indicaciones de la función δ. Si una instancia I queda representada por la cadena $x \in \Omega^*$, diremos que el tamaño de x (o, eventualmente, de I) es n si x tiene n símbolos. Lo notaremos $|x| = n$.

Dado un programa M sobre una Máquina de Turing Determinística, y $x \in \Omega^*$, diremos que M *acepta* a x si, al serle ingresado x como input M alcanza el estado q_Y. El *lenguaje reconocido* por M, $\mathcal{L}_M \subset \Omega^*$, será el formado por todas las cadenas x de símbolos de Ω que son aceptadas por M:

$$\mathcal{L}_M = \{x \in \Omega^* : \quad M \text{ acepta } x\} \tag{7}$$

Diremos que un programa M ***resuelve*** un problema de decisión Π (bajo un esquema $\mathbf{e}$) si para todo input x M alcanza en tiempo finito un estado terminal (q_Y ó q_N), y $\mathcal{L}_M = \mathcal{L}(\Pi, \mathbf{e})$.

Definimos, entonces, la clase de lenguajes:

$$\mathcal{P} = \{\mathcal{L} : \text{ existe un programa } M \text{ sobre una } \texttt{MTD} \text{ tal que } \mathcal{L} = \mathcal{L}_M\} \tag{8}$$

La asociación entre problemas de decisión y lenguajes autoriza el siguiente abuso de notación: diremos que un problema Π pertenece a la clase $\mathcal{P}$ si existe algún esquema de codificación $\mathbf{e}$ tal que el lenguaje asociado a Π, $\mathcal{L}(\Pi, \mathbf{e})$ pertenece a $\mathcal{P}$. Observemos que la convención establecida en 6 nos permite independizarnos del esquema. Puesto que los tamaños de una instancia sobre distintos esquemas *razonables* están vinculados polinomialmente, si para algún esquema *razonable* $\mathbf{e}$, $\mathcal{L}(\Pi, \mathbf{e}) \in \mathcal{P}$, entonces $\mathcal{L}(\Pi, \mathbf{e}') \in \mathcal{P}$ para cualquier otro esquema *razonable* $\mathbf{e}'$.

Por su definición, la clase $\mathcal{P}$ corresponde a los problemas de decisión para los cuales existen algoritmos polinomiales. El objetivo de la siguiente sección es definir una clase análoga para los problemas para los que no se conocen algoritmos eficientes.

2.4. Máquina de Turing No Determinística. Problemas $\mathcal{NP}$

Consideremos el problema `SAT` definido en 2.1. Una instancia I de `SAT` será una cadena de cláusulas sobre un cierto conjunto de variables $\{x_1, ..., x_k\}$. No se conoce ningún algoritmo polinomial capaz de decidir si existe una asignación de valores para estas variables de modo que la expresión dada por I sea verdadera. Observemos, sin embargo, que, *dada* una asignación de valores, es muy sencillo verificar si ésta hace la expresión verdadera o falsa: basta reemplazar cada literal por el valor correspondiente y realizar las operaciones binarias necesarias. En particular: si se tiene una intancia

positiva I de `SAT` y una asignación de valores que hace verdadera la expresión dada por I, existe un algoritmo polinomial que *verifica* este hecho, corroborando que I es positiva.

Esta propiedad de *verificabilidad* polinomial podría generalizarse del siguiente modo: dada un problema Π y una instancia $I \in \mathbf{Y}_\Pi$, existe una estructura E (una asignación de valores en `SAT`, un ciclo en `TSP`, etc.), y un algoritmo polinomial que, a partir de E, corrobora que, efectivamente, $I \in \mathbf{Y}_\Pi$. Gran cantidad de problemas de decisión cumplen con esta propiedad. En el caso del `TSP`, fijada una instancia positiva y un ciclo específico resulta sumamente sencillo corroborar que la suma de las distancias del ciclo es menor o igual que una cota dada. Esta verificación implica apenas tantas sumas como ciudades. En rigor, todos los problemas de decisión derivados de una relajación de un problema de optimización combinatoria lo hacen: la *estructura* E es, en este caso, un elemento particular $s \in S$; para verificar que una instancia I es positiva sólo hace falta calcular $f(s)$.

La clase $\mathcal{NP}$ será definida a través de una formalización de esta idea, y conformorá el universo de problemas de decisión abarcados por la teoría. Para llevar a cabo esta formulación definimos la Máquina de Turing No Determinística.

Una Máquina de Turing No Determinística es una construcción formal similar a la de la Máquina de Turing Determinísitica. Tiene, como ésta última, una cinta infinita divida en casilleros, un dispositivo Controlador de Estados y un Lecto-Grabador asociado a él. A éstos elementos se agregan:

1. Un Dispositivo Adivinador cuya función es, literalmente, *adivinar* la estructura E que hace positiva a una instancia dada; y
2. Un nuevo Lecto-Grabador, asociado al dispositivo de adivinación.

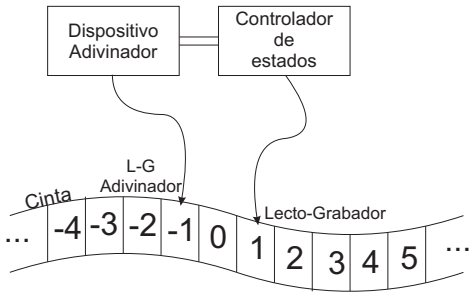


Figura 3: Máquina de Turing No Determinística.

Un programa M sobre una $MTND$ constará de los mismos elementos que formaban un programa en una máquina Determinísitica. Como antes, el primer paso del programa será cargar los datos de una instancia I en los primeros casilleros de la cinta, empezando por el número 1. Ahora bien, una vez ingresada la instancia, el programa pasará por dos fases sucesivas claramente distinguidas:

1. Aquí el Controlador de Estados y su Lecto-Grabador permanecen inactivos, mientras entra en juego el Dispositivo de Adivinación. Éste último *adivina* una estructura E que su correspondiente Lecto-Grabador escribe en la cinta, empezando por el casillero -1 y siguiendo hacia la izquierda. Una vez finalizado este proceso, el Adivinador y su Lecto-Grabador quedan inactivos.

2. Esta fase es idéntica a un programa sobre una Máquina Determinística, cuya función es *verificar*, a partir de la estructura E adivinada en la fase anterior y guardada en los lugares negativos de la cinta, que la instancia almacenada en los lugares a la derecha del 0 está en $\mathbf{Y}_\Pi$.

Queda claro que, a diferencia de la Máquina de Turing Determinística, que esquematizaba el comportamiento real de una computadora, la Máquina de Turing No Determinística es una construcción puramente formal, sin correlato en la realidad. El *Dispositivo de Adivinación* no es más que un artificio que sirve para concretar dentro del programa la existencia de una estructura que haga positiva a la instancia del input y pueda ser luego *verificada* por la parte *normal* del programa.[6]

Análogamente a lo hecho para la Máquina de Turing Determinística, diremos que un programa M sobre una $MTND$ *acepta* a una cadena $x \in \Omega^*$ si para alguna de las infinitas posibles cadenas en Γ^* susceptibles de ser *adivinadas* por el Dispositivo de Adivinación, M alcanza en un tiempo finito el estado q_Y al serle ingresada x como input. Nuevamente, el *lenguaje reconocido* por M es:

$$\mathcal{L}_M = \{x \in \Omega^* : \quad M \text{ acepta a } x\}$$

Definiremos el *tiempo* que necesita M para aceptar x como el mínimo número de operaciones que debe realizar la fase determinística de M antes de llegar a q_Y, tomado sobre todas las posibles estructuras que, al ser adivinadas en la primera fase, llevan a este resultado positivo. Consecuentemente, la *función de complejidad* de M vendrá dada por:

$$\mathcal{T}_M(n) = \text{máx}\left\{\{1\}; \left\{ \begin{array}{c} m : \exists x \in \mathcal{L}_M,\ |x| = n/ \text{ el tiempo necesario} \\ \text{para que } M \text{ acepte a } x \text{ es } m \end{array} \right\}\right\} \tag{9}$$

De acuerdo con esta definición diremos que un programa M es polinomial sobre una Máquina de Turing No Determinística si existe un polinomio p tal que $\mathcal{T}_M = \mathcal{O}(p(n))$.

Observemos que, dado un problema Π, la existencia de un programa M polinomial sobre una $MTND$ que lo resuelva es equivalente a que Π satisfaga la propiedad de

[6] Puede verse otro modo de desarrollar esta idea en [8], Cap. 15, o, más brevemente, en [11]. Allí se define un *certificado suscinto* (*concise certificate*), como la estructura E a partir de la cual puede verificarse la positividad de una instancia. La *concisión* del certificado proviene de la exigencia de que su longitud quede acotada por un polinomio en el tamaño de la instancia. Un problema pertenecerá a $\mathcal{NP}$ si toda instancia positiva I admite un certificado suscinto $c(I)$.

verificabilidad polinomial de la que hablamos más arriba. De este modo, de manera análoga a lo hecho en la sección anterior, definiremos:

$$\mathcal{NP} = \{\mathcal{L} : \exists \text{ un programa polinomial } M \text{ sobre una } MTND \text{ tal que } \mathcal{L}_M = \mathcal{L}\}$$

Nuevamente, aprovechando la asociación entre problemas y lenguajes establecida en 6, nos permitiremos un abuso de notación y diremos que un problema Π pertenece a $\mathcal{NP}$, si $\mathcal{L}(\Pi) \in \mathcal{NP}$.

Con estas definiciones es inmediato que $\mathcal{P} \subseteq \mathcal{NP}$. En efecto, si existe un programa M sobre una MTD que resuelve un problema Π en tiempo polinomial, entonces puede construirse un programa sobre una $MTND$ que verifique que una instancia dada de Π es positiva. Para ello basta con poner M como la segunda parte del programa, ignorando el resultado de la fase adivinatoria. No queda para nada clara, sin embargo, la inclusión opuesta. De hecho, como dijimos anteriormente, se cree que algunos problemas, como `SAT`, `TSP` y `CSP`, son intrínsecamente intratables. La traducción de esta aseveración al lenguaje de clases que hemos introducido es ni más ni menos que: $\mathcal{P} \subsetneq \mathcal{NP}$. Esta es una de las conjeturas abiertas más importantes de la matemática moderna.

2.5. Reducción Polinomial

El concepto de reducción polinomial formaliza la idea de que un problema Π_1 es *más difícil* que otro problema Π_2; o, por mejor decir: que Π_1 es *al menos tan difícil* como Π_2. De este modo, diremos que Π_2 se reduce polinomialmente a Π_1 cuando la existencia de un algortimo polinomial para resolver Π_1 garantice la existencia de un algoritmo polinomial para resolver Π_2. Puesto que nuestra noción formal de algoritmo es la de Máquina de Turing Determinística, y que ésta opera sobre *lenguajes* y no sobre problemas, la definición de reducción polinomial se dará primero para aquellos. Posteriormente, la asociación entre lenguajes y problemas nos permitirá aplicarla directamente a éstos últimos.

Definición 2.2 *Diremos que un lenguaje* $\mathcal{L}_1 \subseteq \Omega_1^*$ se reduce polinomialmente *a otro lenguaje* $\mathcal{L}_2 \subseteq \Omega_2^*$*, si existe una función* $f : \Omega_1^* \to \Omega_2^*$*, computable por una Máquina de Turing Determinísitica* en tiempo polinomial*, de manera que para toda cadena* $x \in \Omega_1^*$*,* $x \in \mathcal{L}_1$ *si y sólo si la cadena* $f(x) \in \mathcal{L}_2$*. Lo notaremos:* $\mathcal{L}_1 \propto \mathcal{L}_2$*.*

El siguiente lema demuestra que nuestra definición responde a la idea que queríamos formalizar:

Lema 2.1 *Si* $\mathcal{L}_1 \propto \mathcal{L}_2$ *entonces* $\mathcal{L}_2 \in \mathcal{P}$ *implica* $\mathcal{L}_1 \in \mathcal{P}$*.*

Demostración: Supongamos que $\mathcal{L}_1 \subseteq \Omega_1^*$ y $\mathcal{L}_2 \subseteq \Omega_2^*$. Si $\mathcal{L}_2 \in \mathcal{P}$, entonces existe un programa M_2 en una MTD que reconoce a $\mathcal{L}_2$ en tiempo polinomial. Llamemos p_2 al polinomio que acota el tiempo que necesita M_2 para efectuar este reconocimiento. Así mismo, sean M_f el programa polinomial que calcula la función f, y p_f el polinomio que acota el tiempo de este cálculo. Entonces: Por otra parte, $|f(x)| \leq p_f(|x|)$.

Ahora bien, el programa M_1 definido sobre Ω_1^* que consiste en aplicar primero f mediante M_f y luego M_2 es un programa que reconoce a $\mathcal{L}_1$, puesto que, por definición de reducción polinomial, $x \in \mathcal{L}_1 \Leftrightarrow f(x) \in \mathcal{L}_2$. Además, el tiempo que necesita M_1 para reconocer una cadena $x \in \mathcal{L}_1$ es menor o igual que $p_f(|x|)+p_2(p_f(|x|))$, con lo cual existe un polinomio $(p_f + p_2 \circ p_f)$ que acota la complejidad de M_1, es decir: $\mathcal{L}_1 \in \mathcal{P}$.□

Corolario 2.1 (de la demostración) *La composición de funciones computables polinomialmente en una MTD es polinomial.*

Nuestra intención es que $\propto$ nos permita comparar problemas para poder afirmar que un problema es tanto o más difícil que otro. Es razonable esperar, entonces, que $\propto$ defina un *orden parcial* sobre $\mathcal{NP}$. Es claro que $\mathcal{L} \propto \mathcal{L} \quad \forall \mathcal{L} \in \mathcal{NP}$, por lo cual $\propto$ resulta reflexiva. El siguiente lema demuestra que es también transitiva.

Lema 2.2 *Si $\mathcal{L}_1 \propto \mathcal{L}_2$ y $\mathcal{L}_2 \propto \mathcal{L}_3$ entonces $\mathcal{L}_1 \propto \mathcal{L}_3$*

Demostración: Sean Ω_1, Ω_2 y Ω_3 los alfabetos sobre los que se definen $\mathcal{L}_1, \mathcal{L}_2$ y $\mathcal{L}_3$ respectivamente. Como $\mathcal{L}_1 \propto \mathcal{L}_2$ existe una transformación polinomial $f_1 : \Omega_1^* \to \Omega_2^*$. Análogamente, existe una $f_2 : \Omega_2^* \to \Omega_3^*$. Definimos $f : \Omega_1^* \to \Omega_3^*$, $f = f_2 \circ f_1$. Dado $x \in \Omega_1$ tenemos que: $x \in \mathcal{L}_1 \Leftrightarrow f_1(x) \in \mathcal{L}_2 \Leftrightarrow f(x) = f_2(f_1(x)) \in \mathcal{L}_3$. Por otra parte, por 2.1 f resulta polinomial. □

Diremos que un problema Π_1 se reduce polinomialmente a a otro problema Π_2 si existen esquemas de codificación $\mathbf{e}_1$ y $\mathbf{e}_2$, para Π_1 y Π_2 respectivamente, de manera que $\mathcal{L}(\Pi_1, \mathbf{e}_1) \propto \mathcal{L}(\Pi_2, \mathbf{e}_2)$. En este caso, trasladaremos la notación y escribiremos $\Pi_1 \propto \Pi_2$.

Nuestro objetivo será hacer uso de la reducción polinomial para comparar problemas específicos dados. Ateniéndonos estrictamente a la definición anterior, para probar que un problema Π_1 se reduce a otro, Π_2, deberíamos hallar, en primer lugar, esquemas de codificación $\mathbf{e}_1$ y $\mathbf{e}_2$ para cada uno de ellos y, luego, una función f que transforme polinomialmente $\mathcal{L}(\Pi_1, \mathbf{e}_1)$ en $\mathcal{L}(\Pi_2, \mathbf{e}_2)$. Este procedimiento puede amenizarse, sin embargo, sin pérdida de generalidad -y con ganancia de claridad-, desembarazándonos de los esquemas de codificiación, y apelando a la asociación directa entre lenguajes y problemas. De este modo, la existencia de una función de reducción f entre $\mathcal{L}(\Pi_1)$ y $\mathcal{L}(\Pi_2)$ es equivalente, en el plano de los problemas, a la existencia de una función $f : \mathbf{D}_{\Pi_1} \to \mathbf{D}_{\Pi_2}$ de complejidad polinomial, de manera tal que una instancia $I \in \mathbf{D}_{\Pi_1}$, pertenece a $\mathbf{Y}_{\Pi_1}$ *si y sólo si* $f(I)$ pertenece a $\mathbf{Y}_{\Pi_2}$. Esta equivalencia es consecuencia de la asociación directa entre los conjuntos de instancias positivas $\mathbf{Y}_{\Pi_i}$ y los lenguajes de aceptación $\mathcal{L}(\Pi_i)$.

Veamos en un ejemplo cómo demostrar en la práctica una reducción polinomial. Para ello definamos:

`3-SAT`

Designamos con este nombre al caso particular del problema `SAT` en que todas las cláusulas involucradas en la expresión contienen exactamente 3 literales. Observemos que `3-SAT` es $\mathcal{NP}$ pues es un caso especial de `SAT`.

Proposición 2.1 *SAT* $\propto$ *3-SAT*

Demostración: Consideremos una instancia I de `SAT` dada por la expresión:

$$\xi = \prod_{i=1}^{N} C_i$$

donde las $C_i \quad i = 1, ..., N$ son cláusulas sobre el conjunto de variables $X = \{x_1, ..., x_n\}$. Para demostrar la proposición deberemos mostrar una instancia I' de `3-SAT` dada por una expresión ζ sobre un conjunto de variables X', que pueda calcularse en tiempo polinomial a partir de ξ y que cumpla con la propiedad de que ξ pueda ser satisfecha por una asignación de valores para las variables de X si y sólo si lo propio sucede con ζ sobre X'.

La demostración es contructiva: examinaremos las cláusulas de ξ y definiremos para cada una de ellas una secuencia ζ_i de cláusulas de tres literales. La concatenación de las ζ_i formará ζ. Para referirnos a los literales (que pueden ser una variable o su negación) utilizamos la letra λ. Dada una C_i puede darse cualquiera de los siguientes casos:

1. Si C_i tiene exactamente tres literales, no hay nada que hacer. Definimos simplemente $\zeta_i = C_i$.

2. Si C_i tiene más de tres literales: $C_i = (\lambda_1 + \lambda_2 + ... + \lambda_k)$, $k > 3$, definimos nuevas variables: $y_1, y_2, .., y_{k-3}$ y:

   $$\zeta_i = (\lambda_1 + \lambda_2 + y_1) \cdot (\bar{y}_1 + \lambda_3 + y_2) \cdot (\bar{y}_2 + \lambda_4 + y_3) \cdot ... \cdot (\bar{y}_{k-3} + \lambda_{k-1} + \lambda_k)$$

   Queda claro que C_i es positiva si y sólo si ζ_i lo es: una asignación de valores que haga a C_i verdadera, deberá poner alguno de los λ_{i_j} =`V`; copiando ese valor en ζ_i pueden tomarse valores para las nuevas variables y de modo que ζ_i sea verdadera. Inversamente, si ζ_i es verdadera, alguna de las x_{i_j} deberá ser `V`, con lo cual C_i es verdadera. Por otro lado, observemos que $k \leq n$ (si $k > n$ C_i resulta trivialmente verdadera y puede descartarse), de modo que la cantidad de variables auxiliares y y, por lo tanto, la longitud de ξ_i' queda acotada por un polinomio en el número de variables n.

3. Si C_i involucra menos de 3 literales entonces:

- si $C_i = (\lambda)$ definimos variables auxiliares z, w, u, v y:

$$\zeta_i = (\lambda + z + w) \cdot (\bar{z} + u + v) \cdot (\bar{z} + \bar{u} + v) \cdot (\bar{z} + u + \bar{v}) \cdot (\bar{z} + \bar{u} + \bar{v}) \\ \cdot (\bar{w} + u + v) \cdot (\bar{w} + \bar{u} + v) \cdot (\bar{w} + u + \bar{v}) \cdot (\bar{w} + \bar{u} + \bar{v})$$

- si $C_i = (\lambda_1 + \lambda_2)$, definimos variables auxiliares z, u, v y:

$$\zeta_i = (\lambda + z + w) \cdot (\bar{z} + u + v) \cdot (\bar{z} + \bar{u} + v) \cdot (\bar{z} + u + \bar{v}) \cdot (\bar{z} + \bar{u} + \bar{v})$$

La construcción de las cláusulas artificiales fuerza a z y w (o a z, en el segundo caso) a ser falsas, con lo cual ζ_i será verdadera si y sólo si C_i es verdadera. Nuevamente, la longitud de ζ_i está acotada por un polinomio en n.

Finalmente:

$$\zeta = \prod_{i=1}^{N} \zeta_i$$

define una instancia I' de `3-SAT`, sobre el conjunto de variables X' formado por X y las variables auxiliares que hayan sido necesarias, que resulta positiva si y sólo si I era una instancia positiva de `SAT`.□

2.6. Problemas $\mathcal{NP} - Completos$. Teorema de Cook

La reducción polinomial nos permite definir en $\mathcal{NP}$ la relación $\sim$ dada por:

$$\mathcal{L}_1 \sim \mathcal{L}_2 \qquad \Leftrightarrow \qquad \mathcal{L}_1 \propto \mathcal{L}_2 \quad \text{y} \quad \mathcal{L}_2 \propto \mathcal{L}_1$$

De este modo $\sim$ es simétrica por definición, mientras que hereda la reflexividad y la transitividad de $\propto$ (ver corol. 2.2). $\sim$ define entonces una relación de equivalencia en $\mathcal{NP}$. Diremos que $\mathcal{L}_1$ es *polinomialmente equivalente* a $\mathcal{L}_2$ si $\mathcal{L}_1 \sim \mathcal{L}_2$. Como siempre, extenderemos nuestra definición diciendo que Π_1 es *polinomialmente equivalente* a Π_2 si $\mathcal{L}(\Pi_1) \sim \mathcal{L}(\Pi_2)$.

Observemos que si dos problemas son polinomialmente equivalentes la existencia de un algoritmo eficiente que resuelva uno cualquiera de ellos implica automáticamente la existencia de un algoritmo eficiente para el otro.

Definición 2.3 *Un lenguaje $\mathcal{L} \in \mathcal{NP}$ es $\mathcal{NP}-$Completo cuando para todo otro lenguaje $\mathcal{L}' \in \mathcal{NP}$ se tiene que $\mathcal{L}' \propto \mathcal{L}$. Más informalmente, $\Pi \in \mathcal{NP}$ es $\mathcal{NP} - Completo$ si para todo otro problema Π' en $\mathcal{NP}$ se tiene que $\Pi' \propto \Pi$.*

Dado que la reducción polinomial nos da una medida comparativa de la *dificultad* de los problemas, los $\mathcal{NP} - Completos$ serán, por definición, los problemas más difíciles en $\mathcal{NP}$. La clase de los problemas $\mathcal{NP} - Completos$ es, pues, la clase de los problemas considerados *intratables*. Al ser todos polinomialmente equivalentes entre sí, la resolución

mediante un algoritmo eficiente de cualquiera de ellos traería aparejada la inmediata resolución de todos los problemas $\mathcal{NP}$. Es decir: si para algún Π $\mathcal{NP} - Completo$, se tuviese que $\Pi \in \mathcal{P}$, entonces resultaría que $\mathcal{P} = \mathcal{NP}$. Como hemos remarcado anteriormente, la resistencia que han mostrado algunos problemas ha llevado a pensar que $\mathcal{P} \neq \mathcal{NP}$. Esta sospecha obliga a suponer que la geografía de la clase $\mathcal{NP}$ es como la que se describe en la figura 4 Si bien el tema excede los límites del presente trabajo, cabe

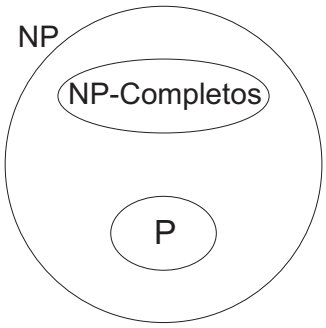


Figura 4: Problable descripción de $\mathcal{NP}$

aclarar que existen problemas $\mathcal{NP}$ *intermedios*, para los que no se conocen algortimos polinomiales (se cree que no están en $\mathcal{P}$) y que, a su vez, no han podido ser clasificados entre los $\mathcal{NP} - Completos$.

La dificultad que han representado los problemas $\mathcal{NP} - Completos$ (e.g.: `SAT`, `TSP`, `CSP`, etc.) en comparación con aquellos que se sabe positivamente que pertencen a $\mathcal{P}$, sumada a la equivalencia que acabamos de exponer entre todos los $\mathcal{NP} - Completos$ obliga a abrigar pocas esperanzas en cuanto a la posibilidad de hallar algoritmos eficientes para los problemas más difíciles. Sin embargo, mientras no se conozca una demostración de que $\mathcal{P} \neq \mathcal{NP}$ (o, eventualmente, de lo contrario), no quedará más remedio que navegar sobre la conjetura, corriendo el riesgo de considerar *intratables* a problemas que no lo son. Tendremos, entoces, resultados del tipo: *si* $\mathcal{P} \neq \mathcal{NP}$, *entonces* Π *es intratable.*

Para poder concluir[7] que un problema Π es intratable deberemos demostrar que Π es $\mathcal{NP} - Completo$. Si nos atuviésemos estrictamente a la definición, deberíamos, para ello, ver que *todo* problema $\mathcal{NP}$ se reduce polinomialmente a Π. Haciendo uso de la equivalencia entre los $\mathcal{NP} - Completos$, el siguiente lema nos permitirá sortear esta dificultad técnica en todos los casos salvo en uno, el primero.

Lema 2.3 *Sean* $\mathcal{L}_1, \mathcal{L}_2 \in \mathcal{NP}$. *Si* $\mathcal{L}_1$ *es* $\mathcal{NP} - Completo$ *y* $\mathcal{L}_1 \propto \mathcal{L}_2$, *entonces* $\mathcal{L}_2$ *es* $\mathcal{NP} - Completo$

Demostración: Sea $\mathcal{L} \in \mathcal{NP}$. Como $\mathcal{L}_1$ es $\mathcal{NP} - Completo$ tenemos que $\mathcal{L} \propto \mathcal{L}_1$. Luego, por la transitividad de $\propto$ (2.2), $\mathcal{L} \propto \mathcal{L}_2$. Y esto vale para cualquier $\mathcal{L} \in \mathcal{NP}$, con lo cual, $\mathcal{L}_2$ es $\mathcal{NP} - Completo$. □

[7] Es decir, conjeturar basados en que $\mathcal{P} \neq \mathcal{NP}$

A partir de este resultado, para demostrar que un problema Π es $\mathcal{NP} - Completo$ deberemos ver:

$$\begin{array}{ll} 1. & \text{Que } \Pi \text{ es } \mathcal{NP} \text{ y,} \\ 2. & \text{Que hay algún } \Pi' \ \mathcal{NP} - Completo \text{ tal que } \Pi' \propto \Pi. \end{array} \tag{10}$$

Obviamente, este tipo de demostración necesita contar con algún Π' a priori y, hasta el momento, no tenemos ninguno. El teorema de Cook demuestra que `SAT` es $\mathcal{NP} - Completo$, y que, por lo tanto, esta clase es no vacía.

Para demostrar el teorema de Cook deberemos ver que *todo* problema en $\mathcal{NP}$ se reduce polinomialmente a `SAT`. Ahora bien: los problemas $\mathcal{NP}$ son aquellos cuyas instancias positivas pueden representarse por lenguajes que son lenguajes de aceptación de alguna Máquina de Turing No Determinística. Demostraremos entonces, que todo programa sobre una $MTND$ puede verse como un programa que verifica que cierta instancia de `SAT` es positiva. Para ello construiremos, a partir de los elementos de una $MTND$ genérica M, una expresión que define una instancia de `SAT` sobre un conjunto apropiado de variables. La longitud de la expresión y, por lo tanto, el tiempo requerido para su construcción, estará acotada por un polinomio sobre el número de elementos de M.

Teorema 2.1 (Cook) *`SAT` es $\mathcal{NP} - Completo$.*

Demostración: En primer lugar, recordemos que `SAT` es $\mathcal{NP}$. Como ya señalamos, dada una instancia I y una asignación de valores específica para las variables bastará con reemplazar cada literal por el valor correspondiente y realizar las operaciones binarias de suma y producto indicadas por la expresión de I. El número de estas operaciones será siempre menor o igual que el número de variables por el número de cláusulas, de modo que estará acotada por un polinomio en el tamaño de I.

Consideremos ahora un programa arbitrario M sobre una Máquina de Turing No Determinística, con componentes: $\Gamma = \{\gamma_0 = b, \gamma_1, ..., \gamma_s\}$ el conjunto de símbolos, Ω el alfabeto, $Q = \{q_0, q_1 = q_Y, q_2 = q_N, q_3, ..., q_r\}$ el conjunto de estados y δ la función de transición; y sea $\mathcal{L}_M \subseteq \Omega^*$ el lenguaje reconocido por M. Sea p un polinomio que acota la función de complejidad de M, $\mathcal{T}_M$. A partir de estos elementos definiremos variables que representarán diferentes *momentos* de M, y que luego vincularemos en cláusulas que describirán su funcionamiento.

Supongamos que ingresamos en el programa un $x \in \Omega^*$ que es aceptado por M y sea $n = |x|$. Para alguna estructura surgida del Dispositivo de Adivinación en la primera fase de M, existirá una rutina de verificación que acepta a x. A lo largo de esta rutina, el Lecto-Grabador no podrá salirse nunca de los casilleros comprendidos entre el $-p(n)$ y el $p(n) + 1$ de la cinta, pues al comienzo de la ejecución se encuentra en 1 y la rutina demanda a lo suma $p(n)$ movimientos.

Para construir una expresión booleana ξ que defina una instancia de `SAT` utilizaremos tres tipos de variables, definidos en la siguiente tabla. Para evitar confusiones manten-

dremos la siguiente convención: el índice i recorrerá el tiempo (los diferentes *pasos* del programa), el j los casilleros de la cinta, el l los estados de Q y el k los símbolos de Γ.

Variable	Rango	Significado
$Q(i,l)$	$0 \leq i \leq p(n)$ $0 \leq l \leq r$	A tiempo i, M está en q_l
$L(i,j)$	$0 \leq i \leq p(n)$ $-p(n) \leq l \leq p(n)+1$	A tiempo i, el Lecto-Grabador está en el casillero j.
$S(i,j,k)$	$0 \leq i \leq p(n)$ $0 \leq l \leq r$ $0 \leq k \leq s$	A tiempo i, el lugar j de la cinta tiene el símbolo γ_k

Como decíamos, nuestro objetivo es retratar el comportamiento de M en una expresión booleana ξ de manera que ξ admita una asignación de valores para sus variables que la haga verdadera *si y sólo si* M alcanza el estado q_Y en un tiempo finito menor igual que $p(n)$. Para ello definiremos seis grupos de cláusulas. Los primeros tres representarán las restricciones generales que rigen el comportamiento de M. El cuarto y el quinto describirán las situaciones inicial y final de M. El sexto grupo, finalmente, corresponderá a las instrucciones de la función δ.

En principio, M se detendrá y dará como resultado SI en cuanto el Controlador de Estados se posicione en q_Y. Para simplificar la construcción de ξ supondremos que M termina a tiempo exactamente $p(n)$. Para esto introduciremos en el mecanismo de M la siguiente modificación: si el Controlador de Estados se encuentra en q_Y a tiempo $t < p(n)$, M no hará nada, pasando directamente al estado $t+1$. Si $t = p(n)$, entonces M se detendrá y dará como resultado: SI. Así mismo, supondremos que la estructura w adivinada en la primera fase de M ocupa *todos* los casilleros de la cinta entre -1 y $-p(n)$. De no ser así, obligaremos al Lecto-Grabador de Adivinación a completar los espacios blancos con símbolos superfluos.

Grupo I

$$\sum_{l=0}^{r} Q(i,l) \qquad 0 \leq i \leq p(n) \tag{11}$$

$$\overline{Q(i,l)} + \overline{Q(i,l')} \qquad \begin{matrix} 0 \leq i \leq p(n) \\ 0 \leq l < l' \leq r \end{matrix} \tag{12}$$

Las $p(n)+1$ cláusulas de (11) indican que en cada tiempo i el Controlador de Estados deberá encontrarse en algún estado, mientras que las $(p(n)+1)(r+1)$ cláusulas de 12 dicen no podrá estar posicionado en más de uno.

Grupo II

$$\sum_{j=-p(n)}^{p(n)+1} L(i,j) \qquad 0 \leq i \leq p(n) \tag{13}$$

$$\overline{L(i,j)} + \overline{L(i,j')} \qquad \begin{array}{c} 0 \leq i \leq p(n) \\ -p(n) \leq j < j' \leq p(n)+1 \end{array} \tag{14}$$

Las $p(n)+1$ cláusulas de (13) obligan al Lecto-Grabador a apuntar, en cada tiempo i a algún casillero entre el $-p(n)$ y el $p(n)+1$. Las $2(p(n)+1)p(n)$ de 14, por su parte, le impiden señalar más de un casillero a la vez.

Grupo III

$$\sum_{k=0}^{s} S(i,j,k) \qquad \begin{array}{c} 0 \leq i \leq p(n) \\ -p(n) \leq j \leq p(n)+1 \end{array} \tag{15}$$

$$\overline{S(i,j,k)} + \overline{S(i,j,k')} \qquad \begin{array}{c} 0 \leq i \leq p(n) \\ -p(n) \leq j < j' \leq p(n)+1 \\ 0 \leq k < k' \leq s \end{array} \tag{16}$$

De manera similar a lo ocurrido con los grupos anteriores, las $2(p(n)+1)^2$ cláusulas de (15) dicen que en cada tiempo i y cada casillero j de la cinta deberá haber algún símbolo de Γ, mientras que las $2(p(n)+1)^2(s+1)$ de (16) prohiben que haya más de un símbolo a la vez en el mismo lugar.

Grupo IV

Si el input viene dado por la cadena de símbolos, $x = \gamma_{k_1}\gamma_{k_2}...\gamma_{k_n}$, lás cláusulas (o productos de cláusulas) unitarias que forman este grupo son:

$$(Q(0,0))(L(0,1))(S(0,0,0)) \tag{17}$$

$$\prod_{j=1}^{n}(S(0,j,k_j)) \tag{18}$$

$$\prod_{j=n+1}^{p(n)+1}(S(0,j,0)) \tag{19}$$

Las primeras tres (17) señalan que en el comienzo de la fase determinística de M el Controlador de Estados se encuentra en q_0, el Lecto-Grabador en

el casillero 1 y que el lugar 0 de la cinta está ocupado por el símbolo b, que separa el input x de la estructura adivinada w. Las demás indican que en los casilleros positivos de la cinta se almacenan los símbolos que describen el input (18), y luego se completa el espacio con b (19). Suman en total $p(n)+4$ cláusulas unitarias.

Grupo V

Está formado símplemente por la cláusula unitaria

$$(Q(p(n)+1,1)) \tag{20}$$

para señalar que en el instante final de la ejecución de M, $p(n)+1$, el Controlador de Estados está en $q_1 = q_Y$.

Grupo VI

Lo veremos en dos partes. El primer subgrupo está formado por $2(p(n)+1)^2(s+1)$ cláusulas de la forma:

$$\overline{S(i,j,k)} + L(i,j) + S(i+1,j,k) \qquad \begin{array}{c} 0 \leq i \leq p(n) \\ -p(n) \leq j \leq p(n) \\ 0 \leq k \leq s \end{array} \tag{21}$$

que garantizan que el símbolo del lugar j no cambie en el paso i si el Lecto-Grabador no apunta a ese casillero en dicho tiempo.

Las restantes cláusulas describen directamente las instrucciones de δ. Supondremos que $\delta(q_l, \gamma_k) = (q_{l'}, \gamma_{k'}, \Delta)$, y si $q_k = q_Y$ o $q_k = q_N$ entonces $\Delta = 0$, $k' = k$ y $l' = l$. Con esta notación, las restantes cláusulas son:

$$\overline{L(i,j)} + \overline{Q(i,l)} + \overline{S(i,j,k)} + L(i+1, j+\Delta) \tag{22}$$

$$\overline{L(i,j)} + \overline{Q(i,l)} + \overline{S(i,j,k)} + Q(i+1,l') \tag{23}$$

$$\overline{L(i,j)} + \overline{Q(i,l)} + \overline{S(i,j,k)} + S(i+1,j,k') \tag{24}$$

con $0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq l \leq r$ y $0 \leq k \leq s$, en todos los casos. Estas cláusulas garantizan que las modificaciones en la posición del Lecto-Grabador (22), en el estado señalado por el Controlador de Estados (23) y en los símbolos que hay en la cinta (24) se realicen según δ. Entre los tres tipos suman $6(p(n)+1)p(n)(s+1)(r+1)$ cláusulas.

La expresión ξ estará formada por la concatenación, vía conjunciones, de todas las cláusulas descriptas anteriormente. Por construcción, es claro que ξ será una instancia *positiva* de `SAT` si y sólo si la cadena x ingresada como input pertenece al lenguaje reconocido por M, $\mathcal{L}_M$.

Para ver que el tiempo necesario para la construcción de ξ está acotado por un polinomio en n bastará verificar que la longitud de ξ está acotada por un tal polinomio. Notemos, primero, que el número total de literales utilizados en ξ (que dará su longitud), es igual a la cantidad c de cláusulas implicadas por el cardinal del conjunto de variables X, $|X|$. De este modo, si f es la función que transforma M en ξ, tendremos que $|f(x)| = c|X|$. Ahora bien, los números $(s+1)$ y $(r+1)$ que dan la cantidad de símbolos de Ω y la cantidad de estados en Q respectivamente, estarán fijos una vez determinada M, por lo cual actuarán como constantes en la construcción de ξ. De este modo, el número de cláusulas de ξ es un $\mathcal{O}(p(n)^2)$. Por otro lado, el conjunto X está formado por las variables $Q(i,l)$, $L(i,j)$ y $S(i,j,k)$, donde $0 \leq i \leq p(n)$, $0 \leq l \leq r$, $-p(n) \leq j \leq p(n)+1$ y $0 \leq k \leq s$, con lo cual $|X| = 2(r+1)(s+1)(p(n)+1)^2 = \mathcal{O}(p(n)^2)$. De esta manera, tenemos que $|f(x)| = c|X| = \mathcal{O}(p(n)^4)$, y entonces la complejidad de f queda acotada por un polinomio en $n = |x|$ y el teorema queda demostrado.$\square$

3. Clasificación del Problema de Corte de Stock

El objeto de este capítulo es llegar a encuadrar el CSP(O) dentro de la teoría que venimos desarrollando. Para ello, enunciaremos primero algunos problemas de decisión y demostraremos, mediante el método enunciado en (10), que pertenecen a la clase de los $\mathcal{NP} - Completos$. Esta secuencia de pequeños teoremas encadenados nos llevará al resultado final de que el problema de Corte de Stock de decisión es también $\mathcal{NP} - Completo$. Más adelante introduciremos la clase de los problemas $\mathcal{NP} - Hard$ que generalizará nuestro concepto de *intratabilidad* a problemas que, como el CSP(O), no son de decisión.

Para poder formalizar correctamente algunos de los problemas intermedios que estudiaremos, daremos antes algunas definiciones básicas de la teoría de grafos que nos servirán también cuando abordemos el algoritmo de Fritsch y Vornberger.

3.1. Grafos

Llamamos *grafo* a un par de conjuntos V y E, de los cuales el primero contiene los *vértices* o *nodos*, mientras que el segundo es el conjunto de las *ramas* o *arcos* y está formado por pares de elementos de V que consideramos conectados entre sí. La notación usual para un grafo es $G = (V, E)$. En la figura 5 puede verse un ejemplo donde :

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$$
$$E = \{(v_1, v_2), (v_1, v_3), (v_2, v_5), (v_2, v_7), (v_3, v_4), (v_3, v_5), (v_4, v_6), (v_5, v_7), (v_6, v_7)\}$$

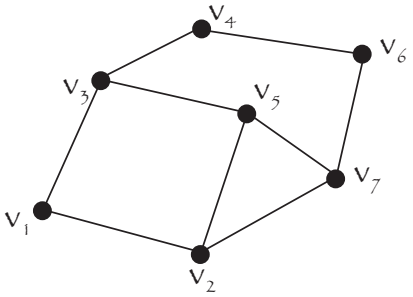


Figura 5: Ejemplo de grafo

Un grafo *completo* es aquel que tiene en E todas las ramas posibles para los nodos dados en V. Notaremos K_n al grafo completo de n vértices. Un *paseo* en un grafo es una sucesión de vértices de V en la que todos los pares de vértices consecutivos están unidos por una rama. E.g.: la sucesión $v_2 - v_5 - v_7 - v_2 - v_1 - v_3$ es un paseo en el grafo de la figura 5. Un *camino* en un grafo es un paseo que no pasa dos veces por el mismo nodo. Un *ciclo* o *circuito* en un grafo es un paseo en el que el nodo inicial coincide con el

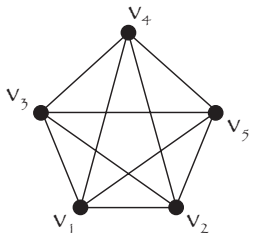


Figura 6: K_5: el grafo completo de 5 vértices

último. Diremos que un ciclo es *simple* cuando los únicos nodos repetidos del ciclo sean el primero y el último. En el grafo de ejemplo, la sucesión v_1, v_3, v_5, v_2, v_1 forma un ciclo simple. Un ciclo se dirá *hamiltoniano* cuando sea simple y pase por todos los nodos de V.

Observemos que, con estas definicioens, el `TSP`, descripto en 1.2, puede verse también como un problema sobre grafos. En este caso, como asumimos que todas las ciudades están interconectadas entre sí, una instancia del problema vendrá dada por un grafo completo K_n y, además, por una función de costo $d : E \to \mathbb{R}$ que asignará a cada rama $(i, j) \in E$ la distancia entre las ciudades i y j. El problema consiste, entonces, en encontrar en K_n un ciclo hamiltoniano de mínimo costo total.

Un grafo se dice *bipartito* cuando el conjunto de vértices puede subdividirse en dos subconjuntos quie llamaremos U y V de manera que los nodos en V sólo estén conectados por ramas con nodos de U, y viceversa. Para este tipo de grafos suele utilizarse la notación: $G = (U \cup V, E)$.

Un *matching* en un grafo bipartito es un subconjunto M de ramas $(u, v) \in E$, con $u \in U$ y $v \in V$ de manera que si $(u', v') \in M$ y (u'', v'') es *otra* rama (distinta) en M, entonces $u' \neq u''$ y $v' \neq v''$. Llamaremos *perfecto* a un matching que involucre a todos los vértices de U. Más generalmente, un matching en un grafo $G = (V, E)$, no necesariamente bipartito, será un subconjunto M de E tomado de manera en M no haya dos ramas distintas que compartan un nodo. Diremos que un matching es *perfecto* en un grafo cualquiera cuando involucra a todos los nodos del grafo si $|V|$ es par, o todos menos uno si $|V|$ es impar. Estas definiciones dan pie para introducir los siguientes problemas:

`MATCHING BIPARTITO (Decisión)`

Sea $G = (U \cup V, E)$ un grafo bipartito. ¿Existe en G un matching M tal que $|M| = |U|$?

`MATCHING (Decisión)`

Sea $G = (V, E)$ un grafo y una constante $B \in \mathbb{N}$. ¿Existe en G un matching M tal que $|M| \geq B$?

`MATCHING PESADO`

Dado un grafo $G = (V, E)$, que podremos suponer completo ($G = K_{|V|}$), y una función $f : E \to \mathbb{R}$ que asigna un *peso* a cada rama de G, se desea encontrar un matching perfecto M de peso total mínimo. Es decir, tal que minimice la expresión:

$$\sum_{e \in M} f(e)$$

Como puede apreciarse, este es un problema de optimización combinatoria.

Los problemas de matching sobre un grafo han sido ampliamente estudiados en sus distintas variantes. En 1965, Edmonds presentó un algoritmo que resuelve el más general de estos problemas, es decir, el que nosotros presentamos como `MATHCHING PESADO`, en $\mathcal{O}(|V|^4)$ iteraciones. Este algoritmo constituye la subrutina principal del método desarrollado por Fritsch y Vornberger para el `CSP`, de modo que será brevemente comentado más adelante. Resulta sencillo demostrar que el problema `MATCHING BIPARTITO Decisión` se reduce polinomialmente al `MATCHING Decisión` (es un caso particular) y que, a su vez, `MATCHING Decisión` se reduce polinomialmente a `MATCHING PESADO`. De este modo, quedaría demostrado que los dos problemas de decisión están en $\mathcal{P}$.

Partiendo de nuestra primera definición, la noción de matching puede generalizarse incrementando el número de conjuntos de nodos. En particular nos interesará el problema con tres conjuntos:

`3D-MATCHING`

Dados tres conjuntos de nodos, U, V y W, con $|U| = |V| = |W|$, y un subconjunto T de $U \times V \times W$, llamaremos *matching tridimensional* a un subconjunto M de T que cumpla que: si $(u, v, w) \in M$ y $(u', v', w') \in M$, $(u, v, w) \neq (u', v', w')$, entonces $u \neq u'$, $v \neq v'$ y $w \neq w'$. El problema consiste en, dado un subconjunto $E \subseteq U \times V \times W$, decidir si existe un matching $M \subseteq E$ tal que $|M| = |U|$. Del mismo modo que en dos dimensiones, un tal matching se llama *perfecto*.

En la siguiente sección veremos que `3D-MATCHING` es $\mathcal{NP} - Completo$ y utilizaremos este resultado para demostrar que `CSP` también lo es.

3.2. Ejemplos de problemas $\mathcal{NP} - Completos$

Lema 3.1 *3D-MATCHING es* $\mathcal{NP} - Completo$

Demostración: `3D-MATCHING` es claramente $\mathcal{NP}$: dada una instancia del problema y un conjunto M de ternas (u, v, w) basta contar para constatar que $|M| = |U|$ y luego comparar los elementos de M con U, V y W para verificar que M es realmente

un matching. Para probar que 3D-MATCHING es completo demostraremos que 3-SAT se reduce a él.

Consideremos una instancia de 3-SAT, dada por las expresión

$$\zeta = \prod_{j=1}^{N} C_j$$

con C_j cláusulas sobre un conjunto de variables $X = \{x_1, ..., x_n\}$. Nuestro objetivo es construir una instancia de 3D-MATCHING. El conjunto U estará formado por una copia de cada literal por cada cláusula:

$$U = \{x_i^j, \bar{x}_i^j : \qquad i = 1, ..., n; j = 1, ..., N\}$$

V y W estarán dados ambos por tres conjuntos análogos:

$$V = \{a_i^j : i = 1, ..., n; j = 1, ..., N\} \cup \{v_j : j = 1, ..., N\} \cup \{c_i^j : i = 1, ..., n-1; j = 1, ..., N\}$$
$$W = \{b_i^j : i = 1, ..., n; j = 1, ..., N\} \cup \{w_j : j = 1, ..., N\} \cup \{d_i^j : i = 1, ..., n-1; j = 1, ..., N\}$$

Los nodos x_i corresponden a las variables de X. La presencia de un x_i en el matching M significará que a tal variable debe asignársele valor verdadero en ζ. Simétricamente, si es la negación $\bar{x}_i$ la que pertenece a M, x_i será falsa. Los nodos a y b están destinados a impedir que el matching incluya a la vez a un nodo x_i y a su negación $\bar{x}_i$. Para ello, ponemos en T las ternas: (a_i^j, b_i^j, x_i^j) y $(a_i^{j+1}, b_i^j, \bar{x}_i^j)$, siempre con $i = 1, ..., n \qquad j = 1, ..., N$ y notando $a_i^{N+1} = a_i^1$.

Las variables v y w sirven para vincular los literales de una misma cláusula. Para ello, pondremos en T, para cada literal λ de la cláusula C_j, la ternas: (v_j, w_j, λ^j) con $j = 1, ..., N$. Las ternas de este tipo que aparezcan en el matching corresponderán también a literales verdaderos en ζ.

Finalmente, como algunos literales habrán quedado sin más conexiones que sus a y b correspondientes y podría resultar, por lo tanto, imposible incorparlos a un matching incluso en el caso de que ζ admitiese una asignación verdadera, agregaremos también para las copia sobrantes de literales, las ternas *de relleno*: $(c_i^j, d_i^j, \lambda^k)$ con $i = 1, ..., n$, $j, k = 1, ..., N$.

De esta manera, por cómo fue construido el conjunto T, queda claro que si existe un 3d-matching perfecto M sobre los conjuntos U, V y W entonces habrá una asignación verdadera para ζ que vendrá, escencialmente, de dar valor verdadero a aquellos literales que hayan sido apareados con un par v_j, w_j. El resto de los literales puede tomar cualquier valor, puesto que con los anteriores tendríamos un literal verdadero por cláusula. A la inversa, si existe una asignación verdadera para ζ, será posible construir un 3d-matching perfecto sobre U, V y W, con lo cual queda demostrado el teorema.□

PARTICIÓN

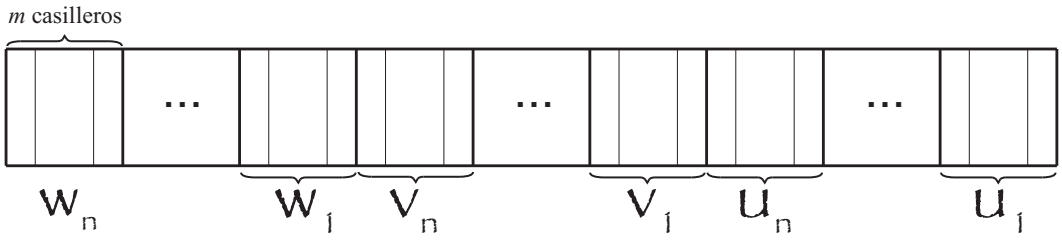
Dado un conjunto de números naturales A se quiere saber si existe un subconjuto $B \subset A$ tal que:

$$\sum_{a \in B} a = \sum_{a \in A \setminus B} a$$

Lema 3.2 *PARTICIÓN es $\mathcal{NP}$ – Completo*

Demostración: Es fácil ver que PARTICIÓN es $\mathcal{NP}$. Dada una instancia del problema y un subconjunto B del conjunto A, basta sumar los elementos en B y los de su complemento para verificar que se trata de una instancia positiva. Para ver que es completo, demostraremos que 3D-MATCHING$\propto$PARTICIÓN.

Sean, entonces, $U = \{u_1, ..., u_n\}$, $V = \{v_1, ..., v_n\}$, $W = \{w_1, ...w_n\}$ y $T = \{t_1, ..., t_N\}$ los conjuntos de nodos y el conjunto de ternas respectivamente. Nuestra instancia de PARTICIÓN tendrá $N+2$ números naturales $a_1, ...a_N, b_1, b_2$. Para cada i, el número a_i será una representación de la terna t_i y vendrá dado por su notación en binario. Los números b_1 y b_2 cumplen una función auxiliar. Notando $m = \log_2(N+1)$, consideraremos, para expresar los a_i, tiras de ceros y unos de longitud $3nm$, formadas por $3n$ clusters que se corresponderán con las variables de U, V y W como se muestra en la figura, divididos en m casilleros cada uno de ellos:



Ahora, dada una terna: $t_i = (u_{\alpha(i)}, v_{\beta(i)}, w_{\gamma(i)})$, le asociaremos el número a_i que en su escritura binaria tiene un 1 en el casillero de la derecha de los clusters correspondientes a $u_{\alpha(i)}$, $v_{\beta(i)}$ y $w_{\gamma(i)}$ y ceros en todos los demás casilleros. Observemos que la subdivisión de cada cluster en m casilleros sirve para que, al sumar algunos de los números a_i, la suma de las componentes correspondientes a U, V y W siga teniendo una representación dentro del mismo conjunto. Es decir: al sumar no se producen *corrimientos* de una zona a la otra. Por otra parte, si existe un matching perfecto M en T, entonces la suma de los a_i correspondientes a los elementos de M será:

$$\alpha = \sum_{i=1}^{n} 2^{m(3n-i)} + \sum_{i=1}^{n} 2^{m(2n-i)} + \sum_{i=1}^{n} 2^{m(n-i)} = \sum_{j=0}^{3n-1} 2^{mj}$$

Los números b_1 y b_2 son definidos de manera que la existencia de una partición del conjunto $A = \{a_1, ..., a_2, b_1, b_2\}$ garantice la existencia de un matching en T. Para ello:

$$b_1 = 2\sum_{i=1}^{N} a_i - \alpha \qquad b_2 = \sum_{i=1}^{N} a_i + \alpha$$

Ahora, si existe un $B \subset A$ tal que $\sum_{a \in B} a = \sum_{a \notin B} a$ entonces b_1 y b_2 no podrán estar ambos en B. Podemos suponer, entonces, que $b_1 \in B$ y $b_2 \notin B$. De esta manera, las ternas asociadas con los elementos de $B \backslash \{b_1\}$ forman un 3d-matching en T. A la inversa, si un tal matching existe, el conjunto formado por los números asociados a sus ternas y el b_1 partirá a A en dos mitades que suman lo mismo.□

BIN PACKING

Dados n items con volúmenes conocidos v_i, $i = 1, ..., n$, ¿Existe una forma de distribuirlos en a lo sumo K recipientes de capacidad V?

Observemos que BIN PACKING no es más que la reducción del CSP de decisión a una dimensión.

Lema 3.3 *BIN PACKING es $\mathcal{NP}$ – Completo.*

Demostración: Es inmediato que BIN PACKING$\in \mathcal{NP}$: dada una distribución de los n items indicando en qué recipiente debe ponerse cada uno, la verificación de que satisface los requerimientos del problema necesitará, por un lado, sumar los items de cada recipiente y comparar el volumen total con V (a lo sumo n sumas y n comparaciones) y, por el otro, comparar el número de recipientes usados con el parámetro K. Es decir, a lo sumo $2n + 1$ operaciones. Para demostrar que es, además, completo, probaremos que PARTICIÓN se reduce polinomialmente a BIN PACKING.

Supongamos que tenemos un conjunto de números naturales $\{a_i\}$, $i = 1, ..., n$, y buscamos un conjunto de índices $I \subset \{1, ..., n\}$ tal que:

$$\sum_{i \in I} a_i = \sum_{i \notin I} a_i$$

Tomemos entonces:

$$V = \frac{1}{2} \sum_{i=1}^{n} a_i$$

Y consideremos la siguiente instancia de BIN PACKING: los volúmenes de los items están dados por los números a_i, el volumen de los contenedores es V y pregunto si existe una forma de distribuir los n items en a lo sumo 2 contenedores. Esta claro que una instancia del primer problema es positiva si y sólo si la correspondiente al segundo lo es. □

Teorema 3.1 *CSP(D) es $\mathcal{NP}$ – Completo.*

Demostración: Dado un patrón de corte, es decir: una secuencia de cortes de guillotina, basta comparar las piezas resultantes con las piezas del pedido para verificar que dicho patrón es efectivamente una solución de la instancia de CSP(D) dada. Por lo tanto, CSP(D)$\in \mathcal{NP}$. Para concluir que es, además, completo, demostraremos que BIN PACKING$\propto$CSP(D).

Dada una instancia de `BIN PACKING` con items $1, 2, ..., n$ de volúmes v_i y K recipientes de volumen V, basta considerar el problema de corte de stock dado por rectángulos $R = (1, V)$ y piezas, $r_i = (1, v_i)$. Resulta evidente que la intancia de `BIN PACKING` es positiva si y sólo si la de `CSP(D)` lo es, pues, de hecho, el problema en ambos casos es exactamente el mismo.□

Hasta aquí hemos demostramos que la versión de decisión del problema que nos interesa pertenece a la clase de los $\mathcal{NP} - Completos$ y que, por lo tanto, a menos que $\mathcal{P} = \mathcal{NP}$, no existirán algoritmos polinomiales que lo resuelvan. Así mismo, sabemos que la versión de optimización es al menos tan difícil como la de decisión y que, por lo tanto, deberemos también considerarla intratable. Esta *intratabilidad*, sin embargo, es hasta ahora informal, puesto que no se encuadra dentro de la teoría de los $\mathcal{NP} - Completos$, dedicada específicamente a los problemas de decisión. En lo que sigue generalizaremos la noción de reducción polinomial para hacerla llegar a problemas que, como el `CSP(O)`, no son de decisión

3.3. Problemas de búsqueda

Definición 3.1 *Diremos que un problema* Π *es un* **problema de búsqueda** *si viene dado por un conjunto de objetos finitos, que llamaremos instancias,* D_Π*; de manera que para cada instancia* $I \in D_\Pi$ *se tiene un conjunto de soluciones de I:* $S_\Pi(I)$*. Diremos que un algoritmo resuelve* Π *si dada* $I \in D_\Pi$*, decide si* $S_\Pi(I) = \emptyset$ *y, en caso contrario, exhibe un elemento* $s \in S_\Pi(I)$*.*

Observemos que todo problema de optimización combinatoria, dado como en (1) y bajo las especificaciones de la definición 1.1 es un problema de búsqueda, donde D_Π viene dado por la descripción de los parámetros del problema que definen el conjunto $\mathcal{S}$ de soluciones factibles de Π. A su vez, para cada instancia I, el conjunto de soluciones $S_\Pi(I)$ será un subconjunto de $\mathcal{S}$, conteniendo los $s \in \mathcal{S}$ que cumplen con la propiedad de minimizar la función f.

Así como los problemas de decisión están asociados con los *lenguajes*, la clase más general de los problemas de búsqueda se asocia con las *relaciones de cadenas*. Dado un alfabeto Ω, llamemos Ω^+ al conjunto de las cadenas no vacías, $\Omega^+ = \Omega^* \setminus \{\varepsilon\}$. Una ***relación de cadenas*** es simplemente una relación $\mathcal{R} \in \Omega^+ \times \Omega^+$.

Como sucedía con los lenguajes, para realizar formalmente la asociación entre problemas de búsqueda y relaciones de cadenas deberemos hacer uso de algún esquema de codificación $\mathbf{e}$, que en este caso deberá brindar una codificación para cada instancia I y una codificación para cada solución s en $S_\Pi(I)$. De este modo, dado Π un problema de búsqueda, definimos:

$$\mathcal{R}(\Pi, \mathbf{e}) = \left\{ (x, y) : \begin{array}{l} x \in \Sigma^+ \text{ es una codificación de una instancia de } \Pi, \\ y \in \Sigma^+ \text{ es una codificación de una solución en } S_\Pi(I) \end{array} \right\}$$

Diremos que una función $f : \Omega^* \to \Omega^*$ *realiza* una relación $\mathcal{R}$, si para cada $x \in \Omega^+$ $f(x) = y$ para algún y tal que $(x, y) \in \mathcal{R}$, o $f(x) = \varepsilon$ si no existe un tal y. Un programa sobre una MTD *resuelve* un problema de búsqueda Π, si la función f_M computada por él realiza la relación $\mathcal{R}(\Pi, \mathbf{e})$, bajo algún esquema de codificación $\mathbf{e}$.

3.4. Reducción de Turing

Más allá de las definiciones formales sobre lenguajes, a la hora de probar que un problema Π_2 se reduce polinomialmente a otro problema Π_1 vimos, esencialmente, que la existencia de un algoritmo polinomial para Π_1 garantiza la existencia de un algoritmo polinomial para Π_2. Más específicamente, asumiendo la existencia de una rutina A que resuelve Π_1, vimos que Π_2 puede resolverse haciendo uso de A cierta cantidad de veces, acotada por un polinómio en el tamaño de Π_2. Si A es polinomial, entonces el programa que lo utiliza para resolver Π_2 también lo es. La noción de reducción de Turing generaliza esta idea, más allá de la polinomialidad de A.

El rol que en el caso de la reducción polinomial jugaba la $MTND$ será cumplido aquí por una Máquina de Turing de Oráculo. Al igual que la $MTND$, la MTO es una construcción puramente formal, que brinda una estructura para la hipotética *existencia* de un algoritmo A.

Una MTO es una Máquina de Turing Determinística a la que se le agrega una nueva cinta, que llamaremos *de oráculo*, con su correspondiente Lecto-Grabador. En el controlador de estados tendremos tres estados distinguidos: q_c, para la *consulta al oráculo*, q_f para finalizar la consulta, y q_p para la finalización total del programa. La función de transición operará de manera análoga a como lo hacía en una MTD, pero sobre ambas cintas. Así: $\delta : (Q \setminus \{q_p, q_c\}) \times \Gamma \times \Omega \to Q \times \Gamma \times \Omega \times \{-1, 1\} \times \{-1, 1\}$.

Si $\delta(q, \gamma_1, \gamma_2) = (q', \gamma_1', \gamma_2', \Delta_1, \Delta_2)$, el Lecto-Grabador de la cinta usual cambiará γ_1 por γ_1' y se moverá según indique Δ_1, mientras que el Lecto-Grabador de oráculo hace lo propio con γ_2, γ_2' y Δ_2. El funcionamiento del *oráculo* se realiza mediante una función $g : \Omega^* \to \Omega^*$, cuyo proceso interno se desconoce. Así, cada vez que el Controlador de Estados se posicione en q_c, el Lecto-Grabador usual se detendrá mientras el Lecto-Grabador de oráculo lee la cadena x escrita en su correspondiente cinta y luego la reemplaza por $g(x)$. Como puede apreciarse, la función g no es más que una representación de la rutina A.

Llamemos M_g a la combinación de una MTD con el oráculo g. Diremos que una relación $\mathcal{R}$ se *reduce Turing* a otra relación $\mathcal{R}'$, si para toda función $g : \Omega^* \to \Omega^*$ que realiza $\mathcal{R}'$ existe una función f, computada por una MTO M_g que realiza $\mathcal{R}$. Pediremos, además, que para cada input x, M_g termine en tiempo acotado por un polinomio sobre $|x|$. Lo notaremos $\mathcal{R} \propto_T \mathcal{R}'$.

Nuevamente, trasladaremos la noción de reducción de Turing directamente a los problemas de búsqueda: fijado algún esquema de codificación $\mathbf{e}$ diremos que Π_2 se reduce Turing a Π_1, si $\mathcal{R}(\Pi_2, \mathbf{e}) \propto_T \mathcal{R}(\Pi_1, \mathbf{e})$.

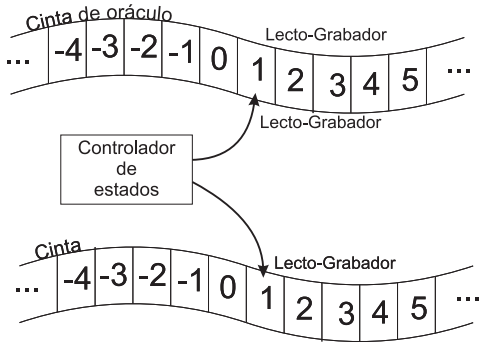


Figura 7: Máquina de Turing de Oráculo.

3.5. Problemas $\mathcal{NP} - Hard$

Observemos, en primer lugar, que todo lenguaje $\mathcal{L}$ puede ser visto como una relación de cadenas. Fijado un símbolo cualquiera $s \in \Omega^*$ tenemos:

$$\mathcal{L} \quad \leftrightarrow \quad \mathcal{R} = \{(x, s) : x \in \mathcal{L}\}$$

Basados en esta asociación, diremos que una relación de cadenas $\mathcal{R}$ es $\mathcal{NP} - Hard$ si existe algún lenguaje $\mathcal{NP} - Completo$ tal que $\mathcal{L} \propto_T \mathcal{R}$, donde $\mathcal{L}$ es visto como una relación. Un problema de búsqueda será $\mathcal{NP} - Hard$ si su correspondiente relación lo es, que es lo mismo que decir: si existe un problema $\mathcal{NP} - Completo$ que se reduce Turing a él. Por definición, queda claro que todo lenguaje (problema) $\mathcal{NP} - Completo$ es $\mathcal{NP} - Hard$. La transitividad de $\propto_T$ es también inmediata.

Hemos cumplido, pues, con nuestro objetivo de extender la noción formal de *intratabilidad* al caso del problema que nos interesa, incluído entre los problemas de búsqueda: la naturaleza de la reducción Turing nos garantiza que ningún problema $\mathcal{NP} - Hard$ podrá ser resuelto en tiempo polinomial a menos que $\mathcal{P} = \mathcal{NP}$. Como sucedía en el caso de la $\mathcal{NP} - completitud$, para demostrar que un problema Π_1 es $\mathcal{NP} - Hard$ nos mantendremos en el plano de los problemas, eludiendo las relaciones de cadenas y las MTO. Buscaremos, entonces, un problema Π_2 al que asociaremos un hipotético algoritmo A que lo resuelva, y veremos que Π_1 puede resolverse haciendo uso de A un número de veces polinomialmente acotado.

Teorema 3.2 *El problema de Corte de Stock de Optimización es $\mathcal{NP} - Hard$.*

Demostración: En primer lugar, sabemos que `CSP(O)` es de búsqueda, porque se ajusta a la definición dada en (1). Para ver que es, además, $\mathcal{NP} - Hard$ demostraremos que se reduce Turing a su versión de decisión.

Supongamos, entonces, que la rutina $\mathcal{A}$ resuelve el `CSP(D)`, y consideremos una instancia I de `CSP` dada por la lista de piezas $r_i = (a_i, l_i)$, $i = 1, ..., n$ y las dimensiones

de las placas de stock, $R = (A, L)$. El tamaño de dicha instancia será el número n de piezas a disponer. Construiremos para `CSP(O)` un algoritmo que fabricará iterativamente instancias de `CSP(D)` que se resolverán mediante $\mathcal{A}$. Observemos que para definir una instancia de `CSP(D)` basta dar una cota $B \in \mathbb{N}$. Entonces:

Inicialmente, tomamos $B = 1$. Si $\mathcal{A}$ aplicado a I con la cota B da una respuesta afirmativa, el algoritmo termina: el número mínimo de placas es B. En caso contrario se pone $B = B + 1$ y se itera el procedimiento.

Tengamos en cuenta que siendo las dimensiones de las r_i menores que las de los rectángulos R, *siempre* podrán disponerse las n piezas en n placas de stock. Esto hace que nuestro algoritmo haga uso de la rutina $\mathcal{A}$ a lo sumo n veces, con lo cual queda probado el resultado.□

Hasta aquí, nos hemos dedicado a la clasificación teórica de nuestro problema. En lo que sigue, y ateniéndonos a las limitaciones que nos plantea la teoría, estudiaremos un método para su resolución. Mostraremos también algunas modificaciones que lo adaptan a nuestro caso particular mejorando su rendimiento y expodremos, finalmente, algunos resultados obtenidos.

4. El Método de Fritsch y Vornberger

Dado que, como demostramos en el teorema 3.2 el problema de Corte de Stock en dos dimensiones pertenece a la clase de los $\mathcal{NP} - Hard$, a menos que $\mathcal{P} = \mathcal{NP}$, no existirán algoritmos polinomiales que lo resuelvan. Debemos resignarnos, por lo tanto, a la implementación de métodos aproximados o heurísticos, que no pretendan alcanzar la solución óptima, sino una *suficientemente buena*.

Por supuesto que la *bondad* de una solución dependerá de muchas circunstancias, no necesariamente inherentes al problema. Lo que buscamos, escencialmente, es un algoritmo que pueda dar, rápidamente, una solución al menos tan buena como la que puede encontrarse *a mano*.

En el caso del problema de Corte de Stock, existe una sencilla verificación que basta para probar que una cierta solución factible es óptima: si el área total de las piezas $\{r_i\}$, $i = 1, ...n$ es mayor que el área total de $k - 1$ placas, entonces está claro que si conseguimos disponerlas en sólo k placas, habremos encontrado una solución óptima. Si hemos utilizado más de k placas, sin embargo, no tenemos condiciones generales que nos permitan decidir si nuestra solución es o no óptima.

El problema de Corte de Stock fue estudiado por Gilmore y Gomory, que lo abordaron usando programación lineal (1961 y 1965). Hertz, en [4], mejoró las técnicas desarrolladas por ellos. Trabajos posteriores, de Whitlock y Christofides (1977), Cani (1979) y Coffman, Garey, Johnson y Trajan (1980), investigaron otros enfoques, proporcionando nuevas técnicas para su resolución. El principal problema de estos algoritmos es que, aún cuando son rápidos, imponen fuertes restricciones al número de piezas que deben cortarse. Otro defecto importante es que, en general, obligan a dar una orientación a priori para las piezas. Por otra parte, métodos con el de Hertz, piden que el número de apariciones de una pieza de pedido én el patrón final de corte no esté acotado, y buscan cubrir con ellas la mayor parte de la superficie de la placa de stock. Resuelven, así, una variante del problema que no es del todo útil en nuestro caso, en el que la cantidad de repeticiones de una pieza no sólo está acotada sino que está determinada a priori. Nuestro objetivo es disponer las piezas dadas sobre el mínimo número posible de placas y, en el mejor de los casos, conseguir que los sobrantes se den en forma de bloques más o menos grandes que puedan ser luego reutilizados.

El algoritmo heurístico aquí estudiado fue desarrollado para la industria del vidrio por Andreas Fristch y Oliver Vornberger en [2] y está basado en el cáculo sucesivo de *matchings* sobre grafos. Admite pedidos de más de cien piezas y contempla la posibilidad de rotarlas. Tiene, sin embargo, sus limitaciones computacionales, que estudiaremos más adelante.

4.1. La idea inicial

Recordemos, en primer lugar, que la solución que esperamos obtener para nuestro problema, planteado en 1.1, vendrá dada por una serie de instrucciones que nos dirán por dónde deben realizarse los cortes (ver Fig.1). Cada corte dividirá la pieza original en dos mitades que serán, a su vez, partidas en dos con el siguiente corte que las afecte. De este modo los cortes finales darán como resultado, o bien dos piezas de pedido, o bien una pieza de pedido y una parte de desperdicio. Llamaremos *corte terminal* a aquel que dé como resultado dos piezas del pedido, eventualmente con un desperdicio adherido. En la Fig.1, los cortes 5, 6 y 8 son *terminales*.

El planteo de Fritsch y Vornberger parte de esta observación y se propone construir el patrón de corte *de atrás para adelante*, es decir: encontrando primero los cortes que serán luego *terminales*. Esto equivale a ir agrupando las piezas del pedido, primero de a pares y luegos en ternas o cuartetos, y así siguiendo hasta tenerlas todas unidas en una misma configuración que no es otra cosa que el patrón de corte final. Estos agrupamientos deben realizarse, por supuesto, teniendo en cuenta las restricciones de largo y ancho totales impuestas por las dimensiones de los rectángulos de stock. En el caso de la industria del vidrio, existe también una restricción para la distancia entre los cortes. En el caso de la madera, esta restricción no existe, pero debe tenerse en cuenta que cada corte consume unos $2,5$mm.

El algoritmo construirá, inicialmente, un grafo completo en el que cada nodo representará una pieza del pedido. A cada rama (u, v) se le asignará un peso que procurará medir el desperdicio ocasionado por el apareamiento entre las piezas representadas por los nodos u y v. Sobre ese grafo se resolverá el problema `MATCHING PESADO` (3.1), obteniendo como resultado $\frac{n}{2}$ parejas de piezas si n es par, o $\frac{n-1}{2}$ parejas y una pieza suelta si n es impar. Estas parejas son tomadas luego como nuevos rectángulos y se reitera el procedimiento.

4.2. Algunas definiciones previas

4.2.1. El input

El input del algoritmo estará dado, por un lado, por las piezas r_i del pedido, es decir, por un listado de los pares (a_i, l_i) de anchos y largos; y, por el otro, por las dimensiones de los rectángulos de stock $R = (A, L)$. A estos datos se agrega una variable binaria que indica si las piezas pueden o no ser rotadas. En una versión un poco más sofisticada, esta variable puede estar asociada a las piezas en lugar de al problema en general. De este modo, tendremos la posibilidad de rotar algunas de ellas, dejando las otras con orientación fija.

4.2.2. La solución - Árboles de Corte

La estructura de la solución, que describimos en la sección anterior, nos permitirá representarla en la forma de un árbol binario. Esto es: un grafo sin ciclos en el que cada nodo está conectado con uno que consideraremos su *padre* y con otros dos que serán sus *hijos*. En la figura puede verse un patrón de corte con el esquema de su correspondiente árbol.

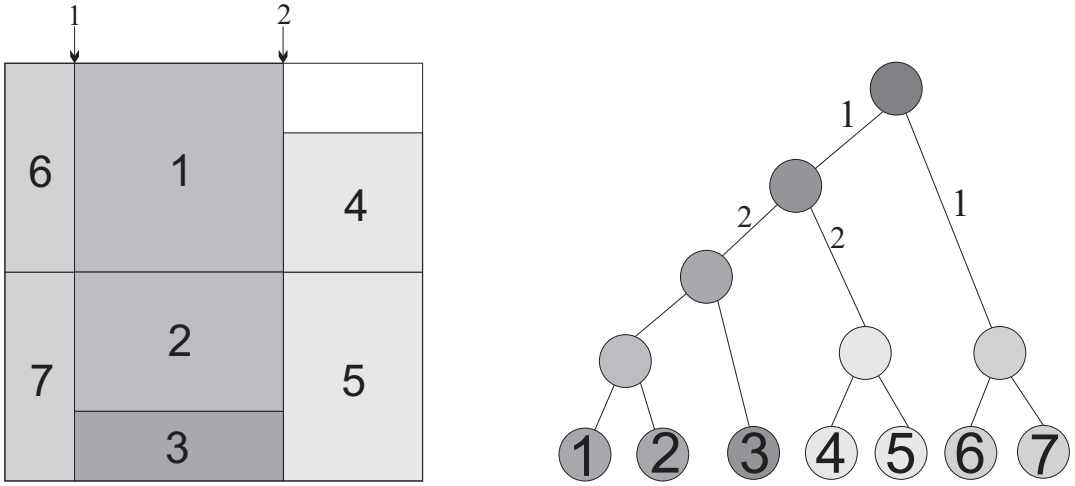


Figura 8: Un patrón de corte con su correspondiente *árbol*.

4.2.3. Meta-Rectángulos

Como advertimos anteriormente, el primer paso del algoritmo formará pares de piezas que irán unidas en el diagrama de corte final. La idea es que este apareamiento dará como resultado nuevos rectángulos que serán tomados, en la siguiente iteración como otras tantas piezas. La dificultad fundamental de este planteo es que la unión de dos piezas **no** da como resultado exactamente un rectángulo.

La unión de dos piezas puede realizarse de varias formas escencialmente distintas: en el caso en que no se admiten rotaciones, la pieza 1 puede ubicarse a la derecha de la pieza 2 (y, por convención, sobre el borde inferior de ésta), o encima de la pieza 2 (y, por convención, sobre el borde izquierdo de ésta); si las piezas pueden rotarse, a estas dos alternativas se agregan otras seis que corresponden a los apareamientos análogos cuando se rota la pieza 2 y no la 1, la 1 y no la 2 y ambas piezas respectivamente. Todas estas situaciones se muestran en la Figura 9.

Un ***metarectángulo*** es una lista de los rectángulos que resultan de la unión de algunas de las piezas de stock. Por ejemplo: el metarectángulo resultante del apareamiento de dos piezas admitiendo rotaciones estará formado por *todos* los rectángulos mostrados en la figura 9. En el caso sin rotaciones, el metarectángulo tendrá sólo los rectángulos

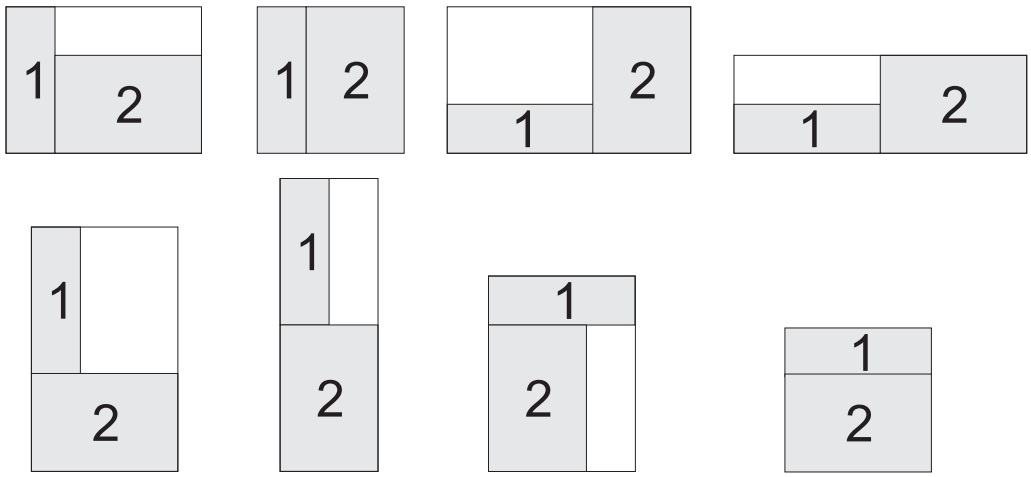


Figura 9: Ocho posibles apareamientos, para el caso en que se admiten rotaciones.

de la primera columna de la figura. Es importante remarcar que un metarectángulo **no** es un réctangulo, y que, por lo tanto, no determina una forma de cortar las piezas que lo conforman. En las instancias finales, el algoritmo deberá *depurar* los metarectángulos, quedándose sólo con los rectángulos que correspondan al patrón de corte que será efectivamente dado como solución.

De aquí en adelante, cuando hablemos de *rectángulos*, nos estaremos refiriendo: o bien a una pieza de stock, o bien a los rectángulos concretos que resultan de la unión de varias piezas (completando los espacios con desperdicio) y que conforman los *meta*rectángulos.

4.2.4. Transversales

Ya mencionamos en 1.1 que la industria del vidrio, para la cual diseñaron su algoritmo Fritsch y Vornberger, trabaja con placas de stock mucho más largas que anchas, lo cual obliga, por la fragilidad del material, a realizar los primeros cortes paralelos al lado más corto. De este modo, la placa original queda dividida en varias planchas del mismo ancho pero de menor longitud, a las que llamaremos transversales. Tanto en la Fig. 1 como en la Fig. 8, los cortes 1 y 2 parten la placa de stock en transversales.

Definiremos, entonces, un *rectángulo transversal*, o simplemente un ***transversal*** como un rectángulo cuyo largo es menor que el largo total de las placa de stock y cuya ancho es aproximadamente el ancho de la placa. En la práctica ambas dimensiones quedan definidas con mayor precisión, por las características de los materiales y de las máquinas que realizan el corte. El largo de un transversal, por ejemplo, puede ser restringido al tercio o el cuarto del rectángulo de stock, con una tolerancia del 5 %, mientras que para el ancho puede fijarse una luz máxima respecto del ancho de la placa.

El algoritmo intentará, a través del matching iterado, construir transversales que luego dejará momentáneamente de lado, retomándolos luego para su disposición final.

Teniendo en cuenta la naturaleza múltiple de los metarectángulos, debemos hacer notar que en un mismo metarectángulo pueden convivir rectángulos transversales con otros que no lo sean. Cuando se detecte un transversal dentro de un metarectángulo, éste será excluido de la siguiente iteración del algoritmo de matching. A su vez, el transversal será marcado como el rectángulo *útil* del metarectángulo.

Diremos que un *metarectángulo* es *transversal* cuando contenga al menos un rectángulo universal. Para evitar confusiones, cuando hablemos de transversales, a secas, nos estaremos refiriendo a rectángulos transversales.

4.2.5. Rectángulos Universales

Al cabo de cada iteración se obtendrá una serie de metarectángulos con la que se fabricará el grafo sobre el que se correrá la iteración siguiente. Un *merarectángulo universal* [8], es un metarectángulo para el cual existe otro que, apareado con él, forma un metarectángulo transversal. Ambos metarectángulos deben estar disponibles al final de una misma iteración. En el algoritmo original se agrega, también, una condición sobre el desperdicio: para que un metarectángulo sea considerado universal, el promedio de los desperdicios de sus rectángulos no podrá exceder el 5 %. En nuestro caso, optamos por eliminar esta restricción, puesto que la relacion entre el tamaño de las piezas y el tamaño de la placa suele hacer imposible el satisfacerla.

En particular, llamaremos rectángulos universales o, más brevemente, ***universales*** a los rectángulos específicos de un metarectángulo que unidos con otro forman un *transversal*.

La construcción de los transversales se realizará en dos etapas: primero se correrá una secuencia de matching iterado, con el objeto de formar metarectángulos universales. Al final de cada iteración se verificarán las posibles uniones de los nuevos metarectángulos. Si de la unión de dos metarectángulos M_1 y M_2 se obtiene un metarectángulo transversal, tanto M_1 como M_2 serán apartados, para ser examinados más adelante. Esta secuencia finaliza una vez que no queden metarectángulos disponibles o cuando todas las uniones entre los existentes excedan las dimensiones de la placa.

En la segunda etapa, se volverán a tomar en cuenta los metarectángulos anteriormente apartados, junto con los que hayan quedado disponibles de la etapa anterior, aunque no fueran universales. Sobre ellos se correrá nuevamente el matching iterado buscando formar transversales.

[8] Si bien en algunos casos, como éste, consideramos que la nomenclatura de Fritsch y Vornberger resulta algo confusa o, en todo caso, no del todo descriptiva, optamos por preservar los nombres que utilizaron en su trabajo, limitándonos a traducirlos.

4.2.6. Funciones de forma - Instrucciones de Corte

Cada uno de los posibles rectángulos incluidos en un metarectángulo puede ser descripto por un vector de cuatro dimensiones, (a, l, o, p), que llamaremos ***instrucción de corte***. La primera coordenada, a, corresponde al ancho total del rectángulo (en las figuras: el alto), mientras que l es el largo . o es una variable binaria, que puede tomar valores en el conjunto $\{\texttt{h},\texttt{v}\}$, indicando la orientación en que se realiza el corte de los *sub*-rectángulos que se unieron para formar el rectángulo en cuestión. La variable p da el punto relativo en que debe realizarse el corte. De este modo, la situación de la figura 10 queda representada por la instrucción de corte $(60, 50, \texttt{h}, 20)$.

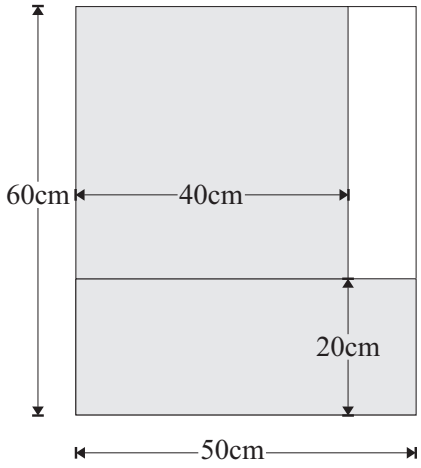


Figura 10: Rectángulo representado por la instrucción $(60, 50, h, 20)$.

Para describir *todos* los rectángulos de un metarectángulo, usamos una *función de forma*[9]. El concepto de las funciones de forma proviene del *floorplan design*, un subproblema del diseño de circuitos.[10]Una función de forma de un (meta)rectángulo r, φ_r, es una función decreciente y lineal a trozos de manera que $\varphi_r(a)$ puede ser interpretada como una cota superior para el largo de r, en función de su ancho a. Dicho de otro modo: dado un metarectángulo conteniendo ciertos rectángulos, su longitud puede ser acotada en función del ancho que se le fije, y eso hace la función de forma. La figura 11 muestra la función de forma de una pieza de demanda.

Como se ve en la figura, la función de forma de una pieza $r = (a, l)$, admitiendo rotaciones, puede darse por un par de instrucciones de corte: $\varphi_r = \{(a, l, \texttt{h}, 0), (l, a, \texttt{h}, 0)\}$. En general, una función de forma de un metarectángulo vendrá dada por una lista de instrucciones de corte que corresponderán a las *esquinas* de la función.

[9]Shape function.

[10]Fritsch y Vornberger citan los trabajos de Otten: *Efficient Floorplan Optimization*; y Stockmayer: *Optimal Orientations of Cells in Slicing Floorplan Design*, a los que no tuvimos acceso. Para precisar algunos aspectos de las *shape functions* recurrimos al trabajo de Wong y Liu, *A new Algorithm for Floorplan Design*, citado en [12]. Allí se las llama *bounding functions* o *bounding curves*, nombre este último que consideramos el más apropiado.

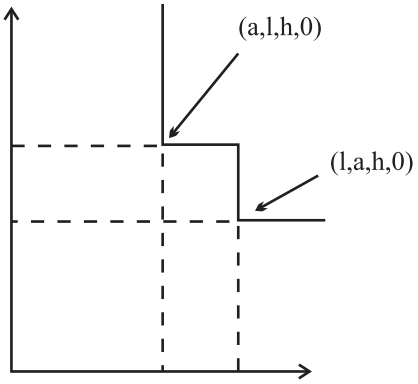


Figura 11: Función de forma de una pieza de pedido.

En la práctica, las funciones de forma son sólo un recurso teórico para el planteo del algortimo. En el momento de la implementación, los metarectángulos serán definidos directamente como una lista de instrucciones de corte y resultarán, por lo tanto, indistinguibles de su función de forma.

La *unión* de dos metarectángulos se realizará mediante la *composición* de sus funciones de corte. Tomemos como ejemplo el apareamiento de dos piezas de stock. Para fijar ideas, supongamos que $r_1 = (2, 5)$, $r_2 = (3, 4)$ y, por lo tanto $\varphi_1 = \{(2, 5, \mathtt{h}, 0), (5, 2, \mathtt{h}, 0)\}$, $\varphi_2 = \{(3, 4, \mathtt{h}, 0), (4, 3, \mathtt{h}, 0)\}$. Tomando una instrucción de corte por cada función, podremos componerlas de dos maneras diferentes, según se ubique r_1 encima o a la izquierda de r_2. Estas dos posibles disposiciones corresponden al sentido del corte horizontal y vertical respectivamente. Tendremos, por lo tanto, dos funciones de composición, una horizontal, σ_h, y una vertical, σ_v. Sean $\gamma_1 = (a_1, l_1, o_1, p_1)$ e $\gamma_2 = (a_2, l_2, o_2, p_2)$ dos instrucciones de corte, entonces:

$$\sigma_h(\gamma_1, \gamma_2) := \big(\,\text{máx}\{a_1, a_2\}, l_1 + l_2, \mathtt{h}, \text{mín}\{l_1, l_2\}\big) \tag{25}$$

$$\sigma_v(\gamma_1, \gamma_2) := \big(a_1 + a_2, \text{máx}\{l_1, l_2\}, \mathtt{h}, \text{mín}\{a_1, a_2\}\big) \tag{26}$$

Siguiendo con nuestro ejemplo y llamando γ_1^i y γ_2^i a las instrucciones de la función φ_i, $(i = 1, 2)$, tenemos:

$$\phi = \sigma(\varphi_1, \varphi_2) = \left\{ \begin{array}{l} \varphi_h = \left\{ \begin{array}{l} \sigma_h(\gamma_1^1, \gamma_1^2) = (3, 9, \mathtt{h}, 4) \\ \sigma_h(\gamma_1^1, \gamma_2^2) = (4, 8, \mathtt{h}, 3) \\ \sigma_h(\gamma_2^1, \gamma_1^2) = (3, 6, \mathtt{h}, 2) \\ \sigma_h(\gamma_2^1, \gamma_2^2) = (5, 5, \mathtt{h}, 2) \end{array} \right. \\ \varphi_v = \left\{ \begin{array}{l} \sigma_v(\gamma_1^1, \gamma_1^2) = (5, 5, \mathtt{v}, 2) \\ \sigma_v(\gamma_1^1, \gamma_2^2) = (6, 5, \mathtt{v}, 2) \\ \sigma_v(\gamma_2^1, \gamma_1^2) = (7, 4, \mathtt{v}, 3) \\ \sigma_v(\gamma_2^1, \gamma_2^2) = (9, 3, \mathtt{v}, 4) \end{array} \right. \end{array} \right.$$

Estas ocho instrucciones de corte se corresponden con las ocho posibles disposiciones de dos piezas, admitiendo rotaciones. Sin embargo, algunas de estas instrucciones pueden

eliminarse del metarectángulo unión: la primera por ejemplo, tiene el mismo ancho que la tercera pero un largo mayor; sería, por lo tanto, un desperdicio manifiesto optar por ella. Lo mismo sucede con la segunda respecto de la tercera, y con la quinta y la sexta, cuyas dimensiones exceden o igualan (ambas) a las de la cuarta. De este modo, estas instrucciones pueden eliminarse de la función composición. El ejemplo quedaría, entonces:

$$\phi = \sigma(\varphi_1, \varphi_2) = \begin{cases} \sigma_h(\gamma_2^1, \gamma_1^2) = (3, 6, \mathtt{h}, 2) \\ \sigma_h(\gamma_2^1, \gamma_2^2) = (5, 5, \mathtt{h}, 2) \\ \sigma_v(\gamma_2^1, \gamma_1^2) = (7, 4, \mathtt{v}, 3) \\ \sigma_v(\gamma_2^1, \gamma_2^2) = (9, 3, \mathtt{v}, 4) \end{cases}$$

La función de composición σ opera, por lo tanto, en dos etapas: en la primera realiza las composiciones σ_h y σ_v de todas las instrucciones de φ_1 con todas las de φ_2, mientras que en la segunda depura el contenido de la composición, eliminando aquellas instrucciones que tengan ancho y largo mayor o igual que alguna otra.

Analíticamente, este proceso de depuración de ϕ equivale a tomar el mínimo entre las composiciones φ_h y φ_v. Esta situación se representa en la figura 12.

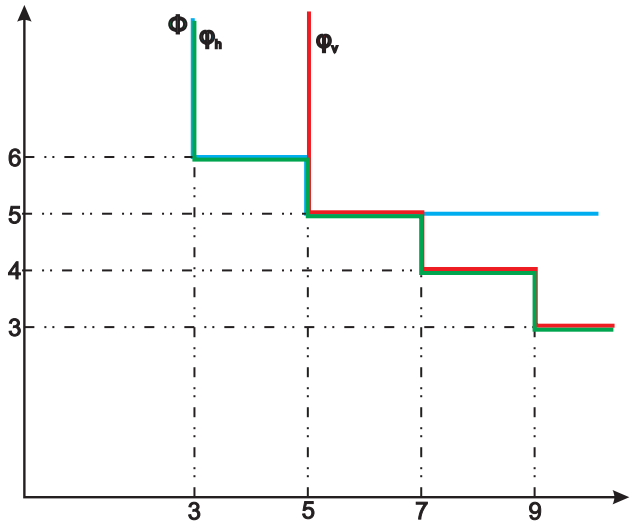


Figura 12: Funciones de composición horizontal y vertical. Y el mínimo entre ambas.

4.2.7. Matching Iterado

En cada iteración, el algoritmo partirá de un conjunto de metarectángulos disponibles y generará, a partir de él, el conjunto de metarectángulos con que se iniciará la próxima iteración. El procedimiento es el siguiente: supongamos que en una cierta iteración i se tienen m metarectángulos, entonces:

1. Se define un grafo completo $G = (V, E)$, con $|V| = m$.
2. A cada rama (u, v) se le asigna un peso proporcional al desperdicio porcentual del metarectángulo resultante de unir los dos (meta)rectángulos representados por los nodos u y v.
3. Se resuelve sobre G el problema `MATCHING PESADO` definido en 3.1.

El problema `MATCHING PESADO` es polinomial. Edmonds desarrolló un algoritmo que encuentra un matching perfecto de mínimo peso sobre un grafo cualquiera, no necesariamente bipartito, en $\mathcal{O}(|V|^4)$ iteraciones[11] Un esquema del algortimo puede consultarse en [8]. En [6], páginas $256 - 258$, se muestra una versión que baja la complejidad a $\mathcal{O}(|V|^3)$. Implementamos este algorimo, en `C++`. En la figura 13 puede verse un ejemplo de un problema de matching de mínimo peso resuelto con nuestra implentación del método de Edmonds.

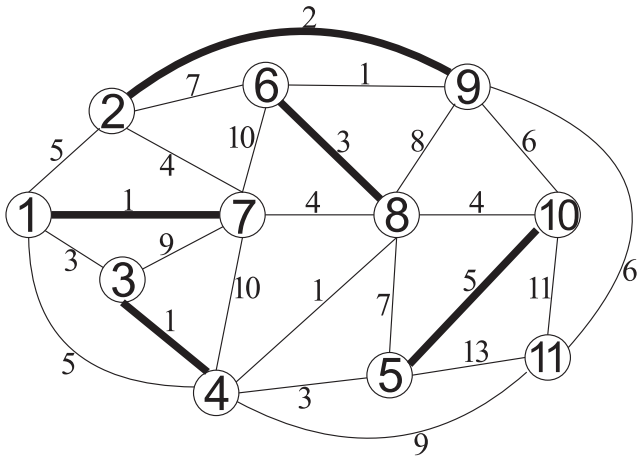


Figura 13: Matching Perfecto de Mínimo Peso.

4.2.8. Bin Packing

El matching iteratdo da como resultado transversales que deben ser luego dispuestos sobre las placas. De este modo pasamos a un problema de `BIN PACKING`, (definido en 3.2), en donde los ítems son los transversales, los volúmenes v_i son sus respectivas longitudes y el volumen máximo V es el largo de la placa de stock. Así, en la práctica, el `CSP` se reduce a su versión unidimensional. Esta reducción parece simplificar el problema y, en algún sentido, lo hace, pero no basta para resolverlo. De hecho, como demostramos mediante el Lema 3.3, `BIN PACKING` es $\mathcal{NP} - Completo$ y, por lo tanto, a menos que $\mathcal{P} = \mathcal{NP}$, no existirán algoritmos eficientes que lo resuelvan.

[11] Edmonds, J. *Matching and Polyhedron with 0-1 Vertices.*

4.3. El Algoritmo

El algoritmo consta de cuatro etapas claramente diferenciadas. Las primeras dos utilizan el matching iterado para la fabricación de transversales. La tercera está destinada al refinamiento de los resultados de las etapas anteriores. La cuarta distribuye los transversales en las placas, resolviendo un problema de `Bin Packing`.

4.3.1. Inicio:

En primer lugar, se fabrican los metarectángulos correspondientes a las piezas del pedido. Esta construcción puede dar como resultado metarectángulos de hasta dos instrucciones de corte en el caso sin rotaciones, o hasta ocho instrucciones en el caso con rotaciones.

4.3.2. Primera Etapa:

Con estos elementos, se define un grafo completo pesado, con tantos nodos como piezas, en el que el peso de cada rama es una representación del desperdicio de la unión de los metarectángulos representados por los nodos.

Sobre este grafo se resuelve el problema `MATCHING PESADO`, mediante el algoritmo de Edmonds. A continuación se realizan las uniones entre los metarectángulos que resulten apareados tras el matching, y se evalúa cuáles de ellos son universales. Éstos son apartados del conjuntos, para ser considerados en la siguiente etapa. Finalmente, se repite el procedimiento con los metarectángulos sobrantes.

El proceso termina cuando ya no hay metarectángulos que no sean universales, o cuando los que quedan no puedan aparearse sin exceder las dimensiones de la placa.

4.3.3. Segunda Etapa:

En esta etapa se toman los metarectángulos dejados por la anterior que, se espera, serán en su mayoría universales[12], y se repite con ellos el matching iterado. De manera análoga a lo que sucedía antes con los universales, cuando se forma un metarectángulo transversal se lo aparta para su consideración posterior. La etapa finaliza cuando no hay más metarectángulos que no sean transversales o cuando los que quedan no pueden ser unidos.

[12] En la sección 5 veremos que no siempre es así.

4.3.4. Tercera Etapa:

La estrategia del matching iterado es *ambiciosa*. Es decir: hace lo que parece más conveniente a corto plazo, apareando las piezas que dan un menor desperdicio en lo inmediato. Esto hace que, a la hora de formar transversales, se introduzcan grandes desperdicios que no pueden mejorarse a partir de los bloques ya formados, pero que podrían haberse evitado considerando las piezas simples. Por ello esta etapa se encarga de realizar un refinado de los metarectángulos procurando disminuir su largo total.

Dados dos transversales[13], representados por sendos árboles de corte, se buscarán recombinaciones de sus subárboles que den como resultados otros dos transversales conteniendo las mismas piezas, pero sumando un largo total menor que los originales. Más adelante describimos con mayor detalle este procedimiento, que queda representado en la figura 14:

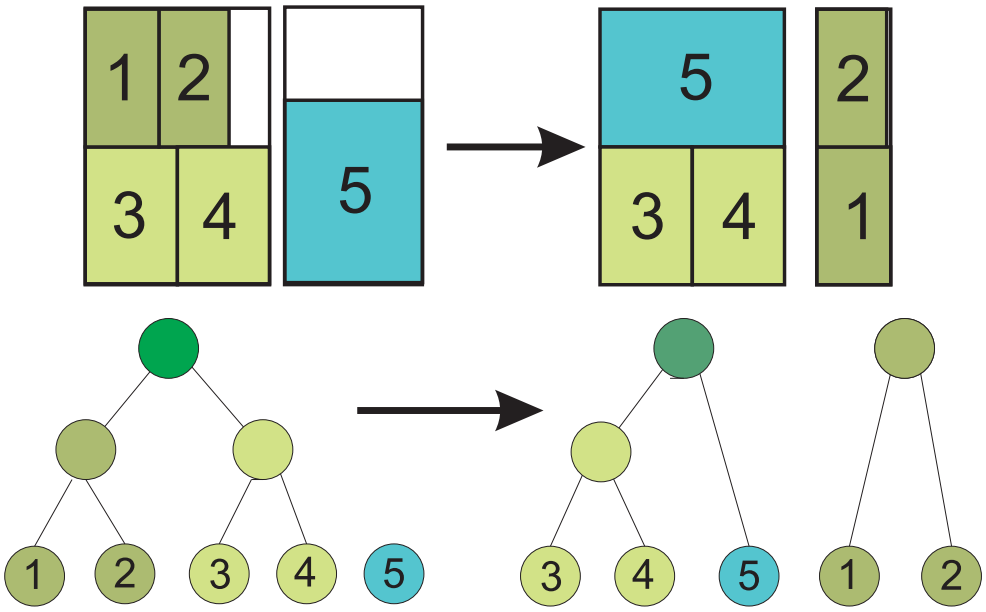


Figura 14: Refinado de Transversales.

4.3.5. Cuarta Etapa:

Finalmente, se disponen los transversales sobre las placas. Como mencionamos anteriormente, esta última etapa corresponde a la resolución del problema $\mathcal{NP} - Completo$ de `BIN PACKING`. Fritsch y Vornberger proponen para ello la utilización del *First Fit Decreasing Algorithm*, a saber:

[13] A partir de aquí el algoritmo toma todos los metarectángulos como transversales, ocupándose sólo del largo.

En primer lugar, se listan las longitudes de los transversales, ordenándolas de mayor a menor. Luego se toma una placa y se recorre la lista, eligiendo el primer bloque que quepa en el espacio que aún queda libre en ella. Cuando una placa ha sido ocupada hasta un punto que no deja lugar para ninguno de los transversales restantes, se toma una placa nueva. Y así hasta disponer todos los transversales.

Veremos que esta estrategia, que constituye el método más simple para el abordaje del `BIN PACKING` resulta muy poco eficiente en el caso de las placas de aglomerado.

5. Implementación y modificaciones

En esta sección comentaremos los aspectos prácticos más interesantes de la implementación del algoritmo. Debe tenerse en cuenta que el trabajo de Fristch y Vornberger sólo esboza los lineamiéntos generales del método, sin precisar un modo específico para la definición de sus estructuras o sus procesos internos. Describiremos, además, algunos defectos importantes que salieron a la luz a la hora de aplicarlo a problemas reales en placas de aglomerado, y las modificaciones que hicimos para mejorar su rendimiento. A costa de no respetar el orden lógico del programa, dejamos para el final los problemas más importantes que encontramos y las variaciones que realizamos para resolverlos.

La implementación se hizo en `C++`, utilizando el entorno de desarrollo GNU `wxDevC++` que incorpora la interfaz gráfica de `wxWidgets` al `DevC++` de Bloodshed. A excepción de la rutina para serialización de objetos *wxSerialize*, que pertenece a Jorgen Bodde y que es de acceso libre en `http://www.xs4all.nl`, todas las rutinas, incluyendo las clases que definen grafos, listas enlazadas, pilas, colas, instrucciones de corte, metarectángulos, etc.; el algoritmo de Edmonds para el problema de matching pesado y las funciones propias del algoritmo definitivo fueron implementadas por nosotros. Los ejemplos se corrieron en un procesador Intel Pentium III.

5.1. Construcción de Metarectángulos

El principal defecto del método de Fristh y Vornberger es que, al considerar los metarectángulos enteros, esto es: todas las formas posibles de juntar cierta cantidad de piezas, el algoritmo se ve obligado a analizar y a guardar en memoria un gran número de intrucciones de corte.

Dadas dos funciones de forma f_1 y f_2 denotemos el número de instrucciones almacenadas en cada una de ellas por $|f_1|$ y $|f_2|$ respectivamente. En las sección 4.2.6 mostramos que la composición de metarectángulos (i.e.: de sus funciones de forma), se realiza componiendo cada una de sus instrucciones de corte horizontal y verticalmente. De este modo, en principio, la composición de f_1 y f_2 deberá guardar un máximo de $2|f_1||f_2|$ instrucciones de corte. El problema reside en que esta cantidad crece exponencialmente a medida que se realiza el matching iterado.

La situación no es, sin embargo, catastrófica, puesto que las dimensiones de las placas de stock limitan el número de apareamientos que es posible realizar. Así, el número de iteraciones quedará controlado, y consecuentemente, también la cantidad de instrucciones de corte a ser almacenadas. Por otro lado, observemos que *siempre* se eliminan algunas instrucciones en el proceso de depuración. Tomemos, por ejemplo, el caso en que las piezas pueden rotarse y compongamos dos funciones de forma simples. Sean, entonces, $f_1 = \{(a_1, l_1), (l_1, a_1)\}$ y $f_2 = \{(a_2, l_2), (l_2, a_2)\}$ (para simplificar la notación eliminamos de las instrucciones las componentes correspondientes a la orientación y al punto de corte). Sin pérdida de generalidad, podemos suponer que $a_1 \leq l_1$ y $a_2 \leq l_2$. La

función composición, sin depurar, será entonces:

$$f = \{(a_1{+}a_2, l_2)(l_1{+}a_2, l_2)(a_1{+}l_2, a_2)(l_1{+}l_2, a_2)(a_2, l_1{+}l_2)(a_2, a_1{+}l_2)(l_2, l_1{+}a_2)(l_2, a_1{+}a_2)\}$$

Ahora bien, en el caso en que $l_1 \leq a_2$, la segunda, la cuarta, la sexta y la octava instrucción pueden eliminarse, pues $a_1 + a_2 \leq l_1 + a_2$ y $a_1 + l_2 \leq l_1 + l_2$. Esta situación está representada en la Fig. 5.1.

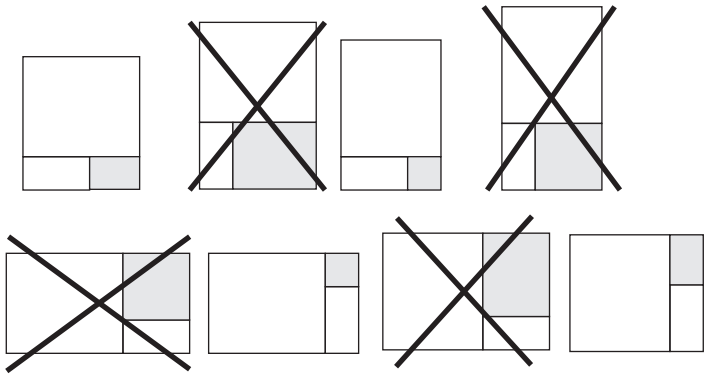


Figura 15: Se eliminan cuatro instrucciones.

Si en cambio, $l_1 \geq a_2$, podemos asumir que $l_2 \leq l_1$ (el caso inverso es análogo), con lo cual se eliminan la segunda y la séptima instrucción (Fig. 5.1), puesto que $a_1{+}a_2 \leq l_1{+}a_2$. Lo mismo sucederá en las siguientes iteraciones, disminuyendo ligeramente la cantidad de instrucciones de corte en cada paso.

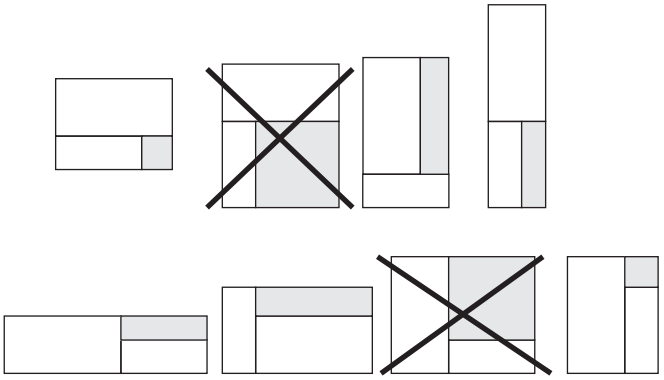


Figura 16: Se eliminan dos instrucciones.

Aún teniendo en cuenta estos hechos, que permiten que el algoritmo funcione, procuramos minimizar el espacio ocupado en memoria, evitando que los metarectángulos almacenen, siquiera transitoriamente, información superflua. Para ello, incorporamos el proceso de depuración al de composición:

En primer lugar, definimos una clase para las instrucciones de corte, de la siguiente manera:

```
class InstruccionDeCorte
{
    private:
        //Generales:
        unsigned int m_iAncho;
        unsigned int m_iLargo;
        bool m_bOrHor;
        unsigned int m_iCorte;
        unsigned int m_iSobrante;
        //Enlistar:
        InstruccionDeCorte* m_pSiguiente;
        InstruccionDeCorte* m_pAnterior;

    public:

        // ... Aquí se definen las funciones generales para el cargado y
            // lectura de los datos, y además:

        InstruccionDeCorte* Siguiente() {return m_pSiguiente;};
        InstruccionDeCorte* Anterior() {return m_pAnterior;};
        void FijarSiguiente(InstruccionDeCorte* siguiente)
        void FijarAnterior(InstruccionDeCorte* anterior)
};
```

Las variables `m_iAncho`, `m_iLargo`, `m_bOrHor` y `m_iCorte`, corresponden a las cuatro componentes de la instrucción de corte tal como la definimos. `m_iSobrante` registra el área total desperdiciada en la configuración representada por la instrucción. Este dato es utiilzado para el cálculo de los pesos del grafo. Las variables que aquí nos interesan son los punteros `m_pSiguiente` y `m_pAnterior`, que apuntarán a otros objetos de tipo `InstruccionDeCorte`. Las funciones `Siguiente()`, `Anterior()`, `FijarSiguiente` y `FijarAnterior` devuelven y almacenan, respectivamente, las direcciones de estos otros objetos. Con estos agregados, nuestra instrucción de corte pasa a ser un elemento de una lista doblemente enlazada. Tal lista será un metarectángulo, para el cual definimos:

```
class MetaRectangulo
{
    private:

        InstruccionDeCorte* m_pPrimera;
        InstruccionDeCorte* m_pUltima;
```

```
        unsigned int m_iNumero;
        unsigned int m_iItem_1;
        unsigned int m_iItem_2;

        unsigned int m_iSobrante;
        unsigned int m_iAreaReal;
        unsigned int m_iCantidad;

        bool m_bTraverse;

    public:
        // ... Funciones de ingreso y lectura de datos, y además:

        bool Agregar(InstruccionDeCorte& dato);
        void Aislar(InstruccionDeCorte* pThis);

        InstruccionDeCorte* Primera() {return m_pPrimera;};
        InstruccionDeCorte* Ultima() {return m_pUltima;};

};
```

Nuevamente, las variables numéricas son los datos concretos del metarectángulo: `m_iNumero` es el número que se le asigna al metarectángulo en cuestión, para poder luego hacer reperencia a él. `m_iItem_1` y `m_iItem_2` son los números correspondientes a los *hijos* del metarectangulos: es decir, a los metarectángulos que se unieron para formarlo. `m_iCantidad` es la cantidad de instrucciones que contiene, mientras que `m_iSobrante` es la suma de los sobrantes de todas ellas. Finalmente `m_iAreaReal` es la suma de las áreas de las piezas que forman el metarectángulo.

Así, el metarectángulo propiamente dicho es una lista que comienza en el puntero `m_pPrimera` (primera instrucción de corte), y termina en `m_iUltima`. Estos dos punteros son las únicas vías de acceso a las instrucciones contenidas en el metarectángulo. Si se desea encontrar una instrucción de corte en particular, debe definirse un puntero lector que comience dirigido a `m_pPrimera` (o a `m_pUltima`), y hacer uso de los punteros `m_pSiguiente` o `m_pAnterior` de cada instrucción de corte para ir avanzando o retrocediendo a través de la lista.

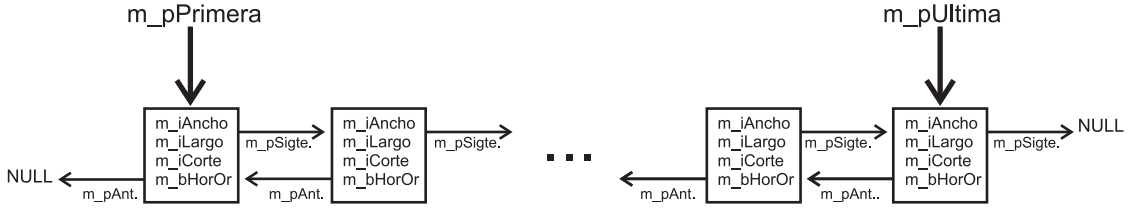


Figura 17: Un metarectángulo es una lista enlazada.

En la Figura 5.1 se esquematiza un metarectángulo, con sus punteros señalando el comienzo y final de la lista, que se enlaza a través de los punteros propios de los objetos de tipo `InstruccionDeCorte`.

Lo importante a los fines de la optimización de memoria se encuentra en la función `Agregar(InstruccionDeCorte&)`. Esta función recibe como parámetro una nueva instrucción de corte y la agrega a la lista, pero de manera que ésta quede ordenada de menor a mayor por los anchos de las instrucciones. Así, el procedimiento para agregar una nueva instrucción es:

1. Se reserva espacio en memoria para una nueva instrucción, con los datos especificados por el parámetro.
2. Se genera un lector que se posiciona en la instrucción señalada por `m_pPrimera`:
   - Si `m_pPrimera` es nula, entonces la lista está vacía: se pone `m_pPrimera` igual a la nueva instrucción y `m_iCantidad`= 1.
   - Si hay una primera instrucción, entonces se compara el ancho de la nueva instrucción con el de aquella señalada por el lector. Mientras el ancho de la nueva sea mayor, el lector avanza a lo largo de la lista. A su vez, en cada paso se compara también el largo de las instrucciones. Si tanto el ancho como el largo de la nueva son mayores (o iguales) que los de una instrucción preexistente, entonces la nueva es descartada y se da por finalizado el proceso.
   - Si la nueva no se descarta el lector alcanzará un punto en que sucede una de las siguientes dos posibilidades:
     - La lista termina: la nueva instrucción se agrega en la cola, y se redefine el puntero `m_pUltima` para que la señale.
     - Se encuentra una instrucción con ancho mayor que la nueva: La nueva se inserta detrás de ella.
3. Si la nueva instrucción fue insertada, entonces sabemos que todas las instrucciones que le siguen tienen ancho mayor que el de ella. En tal caso, el lector continúa recorriendo la lista hasta el final, comparando en cada paso el largo de la instrucción a la que apunta con el de la recién ingresada. Si la nueva tuviese largo menor, entonces la señalada por el lector es eliminada, puesto que excede a la nueva en ambas dimensiones. De este modo, se depura la lista *antes* de continuar agregando instrucciones.

Haciendo uso de esta rutina, entonces, la composición de dos metarectángulos se realiza según el siguiente esquema:

1. Se toma una instrucción `Inst_1` del primer metarectángulo (inicialmente, la primera) y otra, `Inst_2`, del segundo.

2. Se realiza la composición horizontal de ambas, y se pone el resultado en el nuevo metarectángulo destinado a almacenar la composición, haciendo uso de la rutina `Agregar`.

3. Se repite el procedimiento con la composición vertical.

4. Se renuevan los valores de `Inst_1` y `Inst_2`, para que señalen a otras instrucciones de cada uno de las metarectángulos que se están componiendo, y se reitera el procedimiento. La composición finaliza cuando ya han sido estudiados todos los posibles pares de instrucciones.

Además de permitir realizar la depuración al mismo tiempo que se van agregando instrucciones al metarectángulo, el definirlos como listas *ordenadas* facilita también la búsqueda de instrucciones en su interior. Observemos que así como el orden de los anchos de las instrucciones de la lista es creciente, el de los largos resultará *decreciente*: si hubiese una instrucción con un largo mayor que el de la anterior, siendo mayor también su ancho, sería eliminada. De este modo, una vez construidos los transversales, la instrucción que perdurará de un metarectángulo transversal será la última, puesto que será la de menor largo.

5.2. El output

La rutina `Aislar(InstruccionDeCorte*)` de la clase `MetaRectangulo` recibe como parámetro la instrucción del metarectángulo que será efectivamente utilizada en el patrón final de corte, para resguardarla, mientras elimina todas las demás.

Así, al finalizar el algoritmo, se recorren todos los metarectángulos generados, identificando en cada uno la instrucción de corte que será utilizada, y eliminando las restantes.

Observemos que a través de las variables `m_iItem_1` y `m_iItem_2` hemos resguardado la información necesaria para reconstruir el árbol de corte. Una vez terminado el algoritmo, quedarán algunos metarectángulos marcados como *activos* que serán los que correspondan a los patrones de corte finales (uno por placa).

Veamos, entonces, el aspecto de la solución arrojada por el algoritmo para un pequeño ejemplo de cuatro piezas:

```
M(0)=0+0|35496|
232 x 153 |h0|0
--------------------

M(1)=1+1|16625|
125 x 133 |h0|0
--------------------
```

```
M(2)=2+2|16995|
165 x 103 |h0|0
--------------------

M(3)=3+3|19468|
157 x 124 |h0|0
--------------------

M(4)=0+2|52491|
397 x 153 |h165|8250
--------------------

M(5)=1+3|36093|
282 x 133 |h125|1413
--------------------

M(6)=4+5|88584|*
679 x 153 |h282|5640
--------------------
```

El metarectángulo (6), marcado con un $*$ es el único *activo*, es decir: el único terminal. Sus dimensiones son 679×153 mm, y está formado por los metarectángulos (4) y (5). Para obtener estos hijos, (6) debe ser seccionado con un corte horizontal a los 282 mm. Se nos informa además, el area total utilizada (88584mm^2) y el desperdicio interno (5640mm^2).

Para reconstruir la solución hay que desandar el camino: si cortamos a (6) horizontalmente a los 282mm. obtendremos dos partes: una de 282×153 mm. y otra de 397×153 mm. La primera de ellas corresponde al metarectángulo (5) (con un sobrante a lo largo de 20mm.) y la segunda al (4). El (5), a su vez, se corta también horizontalmente, a los 125mm. dando como resultado las piezas (1) y (3). Lo mismo sucede con (4), a los 165mm. obteniéndose las piezas (0) y (1).

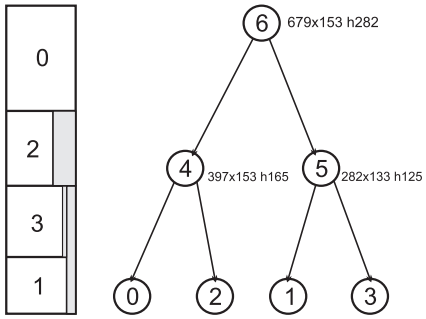


Figura 18: Diagrama y árbol de corte.

5.3. Matching

La primera parte del algoritmo, que realiza el matching iterado sobre los grafos fabricados a partir de los metarectángulos, deja poco que comentar. El método utilizado es el de Edmonds, descripto en [8] y en [6]. Hacemos notar, sin embargo, que Fristch y Vornberger aplican un algoritmo de matching de *máximo* peso. Ésto los obliga a definir un peso para las ramas inversamente proporcional al desperdicio ocasionado por el apareamiento de los (meta)rectángulos correspondientes. En nuestro caso, utilizamos el algoritmo que resuelve el problema de matching de *mínimo* peso total, con lo cual el peso de una rama es, directamente, el desperdicio promedio de la unión.

5.4. Refinado de transversales

Como explicamos en 4.3.4, una vez que se formaron los metarectángulos transversales en la segunda etapa, es necesario refinarlos, para minimizar los desperdicios acumulados a causa del apareamiento de bloques demasiado grandes en las instancias anteriores. En su trabajo, Fristch y Vornberger son bastante escuetos al referirse a esta fase del algoritmo, limitándose a decir que se trata de permutar subárboles. Evidentemente, un estudio exhaustivo de todos los pares de subárboles sería casi un suscedáneo del estudio exhaustivo de todos los posibles patrones de corte, cuyo exhorbitante número hace a la intratabilidad del problema. Es necesario, por lo tanto, establecer criterios de refinación limitados en el grado de exploración de los árboles y, sin embargo, lo más eficacez que sea posible. La puesta en práctica de esta fase requirió progresivos reajustes, basados en los intentos de resolución de ejemplos reales, a menudo insatisfactorios. Finalmente, implementamos cuatro rutinas distintas, que realizan el refinado en diferentes niveles. La más costosa de ellas llega a considerar *nietos* de los transversales en cuestión. No es, sin embargo, la más útil.

5.4.1. Refinado en un nivel:

El caso más sencillo es el que cruza los hijos de dos transversales. Dados `T1` y `T2` transversales, formados por la combinación de los metarectángulos `1h1` y `1h2`, y `2h1` y `2h2` respectivamente, se evalúa si los largos sumados de los metarectángulos resultantes de unir `1h1` con `2h1` y `2h2` con `1h2` es menor que la de `T1` y `T2`. En tal caso, `T1` y `T2` son reemplazados.

En importante observar que este criterio, por si sólo, es de poca ayuda, puesto que, en general, el hecho de ocupar un largo menor –manteniendo el ancho acotado por el ancho de la placa–, implicará también un desperdicio menor. Pero si el desperdicio de los transversales permutados es menor que el de los preexistentes, entonces éstos nunca hubiesen aparecido, puesto que el algoritmo de matching da efectivamente el matching óptimo. De este modo, en una amplia gama de ejemplos, esta refinación aplicado aisladamente no realizará ninguna modificación.

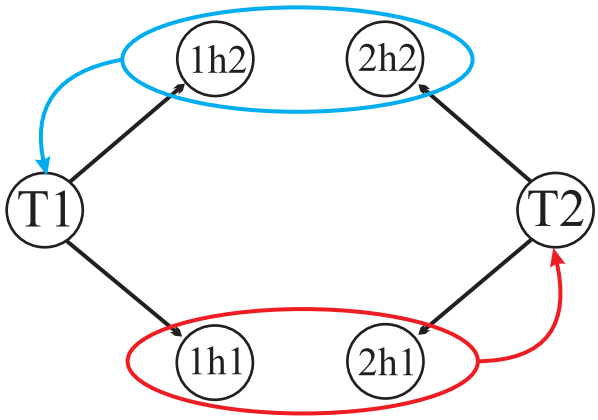


Figura 19: Refinado simple. Se baja un nivel.

5.4.2. Refinado en dos niveles:

Una versión más sofisticada de la misma idea consiste en bajar un nivel más en los árboles correspondientes a los transversales, realizando entonces el mismo procedimiento, en dos pasos: primero se realizan las cruzas entre los nietos para obtener nuevos hijos, y luego se aparean estos nuevos hijos para obtener los nuevos transversales.

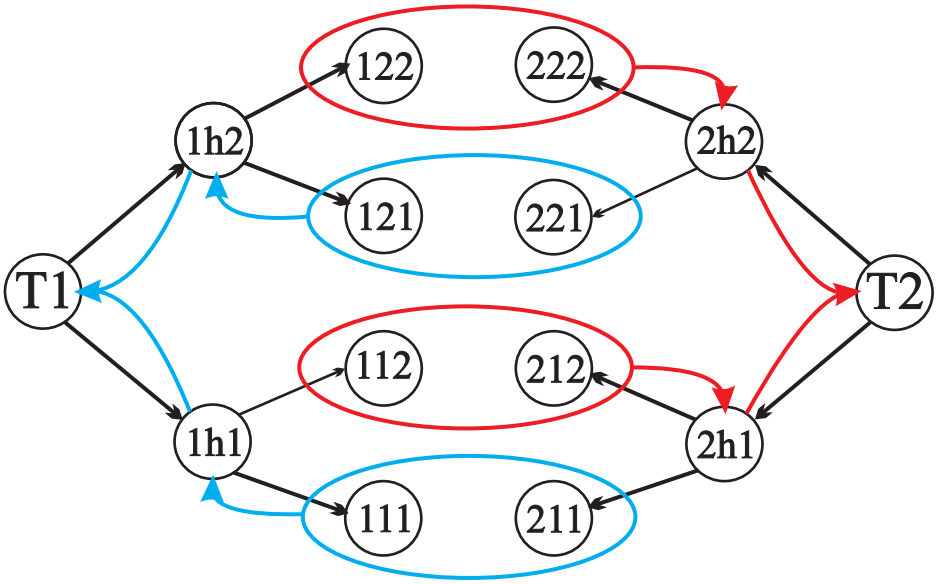


Figura 20: Refinado en profundidad. Se bajan dos niveles.

En la figura puede observarse cómo crece el número de operaciones a realizar, además de la cantidad de metarectángulos auxiliares que deben almacenarse para poder realizar la comparación con los preexistentes.

5.4.3. Refinado desbalanceado:

El matching iterado da como resultado transversales parejos: dado un pedido de n piezas, en la primera iteración se formarán $\frac{n}{2}$ metarectángulos de 2 piezas cada uno, en la siguiente $\frac{n}{4}$ metarectángulos de 4 piezas y así siguiendo. Esta homogeneidad sólo es ligeramente descompensada al apartar metarectángulos universales, o a causa de la imparidad de los bloques examinados en una iteración. De cualquier manera, al finalizar la segunda fase del algoritmo, todos los metarectángulos tendrán aproximadamente la misma cantidad de piezas. Los criterios previos preservan esta situación, que puede resultar costosa por su acumulación de *huecos* o sobrantes que podrían completarse utilizando o bien piezas sueltas o bien agrupaciones de pocas piezas tomadas de otro transversal. Para romper esta estructura, implementamos un tercer método, que baja dos niveles en un transversal y uno en otro.

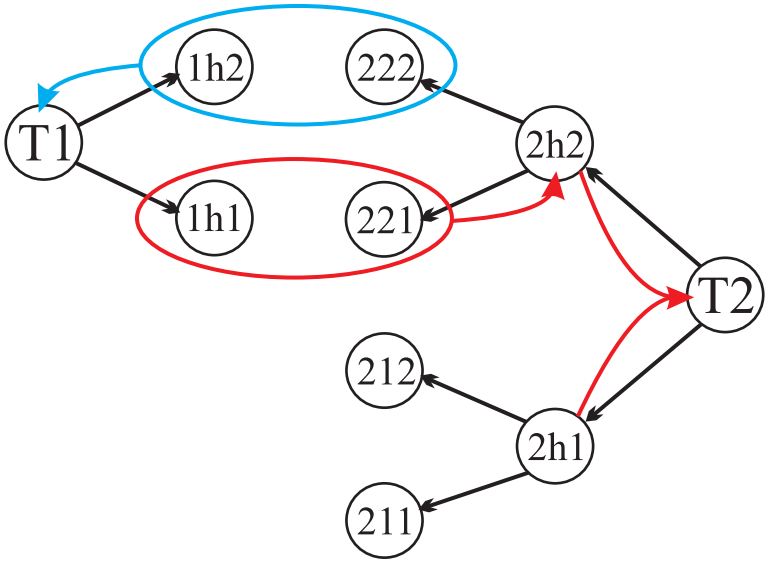


Figura 21: Refinado Desparejo. Se bajan uno y dos niveles.

Siguiendo con la notación de la figura 21, observemos que `1h1` y `2h2` tendrían en principio, el mismo nivel en el arbol de corte general. En otras palabras: al comenzar la etapa de refinamiento, cabe esperar que `1h1` y `2h2` contengan la misma cantidad de piezas. De este modo, al realizar la permutación entre `1h1` y `222`, `2h2` –y, por lo tanto, `T2`– pasará a tener un mayor número de piezas que antes. Simétricamente, la cantidad de piezas en `T1` disminuirá. Así se rompe el equilibrio ocasionado por la iteración del matching.

En la práctica hemos podido constatar que esta última rutina de refinado es particularmente útil, no sólo por la longitud que gana por si misma, sino también porque al quebrar la estructura homogénea de los transversales abre la posibilidad de realizar nuevas permutaciones mediante los primeros dos métodos, que no eran de utilidad en un comienzo.

Los resultados obtenidos con estos tres primeros criterios fueron satisfactorios. Sin embargo, con el objeto de mejorar algo más el rendimiento del algoritmo, implementamos una última rutina. Dado que este cuarto método, que se corresponde con el refinado desbalanceado de un nivel descripto más adelante, fue pensado, originalmente, para la construcción de transversales, dejamos su exposición para la sección 5.5.

En nuestra implementación, las rutinas de refinamiento son aplicadas en forma sucesiva e iterativa. En cuanto se descubre una permutación ventajosa, se realiza, pasando al siguiente criterio. Siempre que se hayan hecho modificaciones en los transversales se renueva el ciclo. La fase de refinado se termina cuando ninguno de los cuatro métodos puede disminuir el largo total de los transversales.

5.5. Construcción de Transversales

Las placas de vidrio para las que fue desarrollado el método son de $2,3 \times 6$ mts. Esta relación largo ancho favorece la fabricación de transversales: cabe esperar que las piezas se agrupen ocupando un ancho y un largo parejos, –particularmente en el caso en que las piezas pueden rotarse, como sucede en el vidrio–. Por lo tanto, los metarectángulos se acercarán al ancho total de la placa mucho antes que al largo. De esta manera, la separación de rectángulos universales y el favorecimiento de la construcción de transversales no hace más que acentuar una tendencia natural del matching iterado, procurando obtener bloques no demasiado largos que puedan ser cortados sin romper la placa y facilitando, a la vez, su posterior disposición en la etapa de `BIN PACKING`.

Por otra parte, la relación entre el tamaño de la placa y el de las piezas será también grande. Es posible, por lo tanto, definir un transversal como un rectángulo de alrededor del cuarto o el tercio del largo total de la placa.

El caso del aglomerado presenta diferencias importantes, que hacen que una implementación del método al pié de la letra desaproveche el espacio notablemente. La placa de melamina es de $1,83 \times 2,6$ mts. Tanto la relación entre el largo y el ancho de la placa como la relación entre el tamaño de la placa y el tamaño de las piezas resultan, entonces, mucho menores que en el vidrio. Si nuestro objetivo fuese formar transversales de un tercio del largo total, nos sería imposible manejar piezas de un metro, habituales en los ejemplos. La cota para la longitud de los transversales deberá ser, entonces, más generosa. En nuestro caso, optamos por un criterio dinámico, que establece el largo máximo de los transversales de acuerdo al tamaño de las piezas, garantizando así que todas ellas puedan ser tomadas en cuenta.

La siguiente dificultad que plantea la diferencia en las dimensiones de las placas tiene que ver con la construcción misma de los transversales. En el ejemplo expuesto en la sección 6 hay varias piezas de $29,8 \times 80$ cm. En las primeras etapas, la política miope del matching iterado agrupa estas piezas por pares y luego por cuartetos. Estos apareamientos son muy beneficiosos a corto plazo, puesto que los conglomerados así formados incluyen muchas piezas, sin ningún desperdicio. Sin embargo, los bloques

resultantes, de $119,2 \times 80$ cm. son imposibles de aparear: estos metarectángulos no sólo no son universales (no pueden formar transversales junto con otros), sino que no admiten apareamiento alguno. De este modo, al finalizar la segunda etapa, se tienen varios metarectángulos que *pasan por* transversales, sin serlo. De hecho, como puede apreciarse en la figura 22, acarrean un importante desperdicio, dejando una luz de más de 60 cm. a lo ancho.

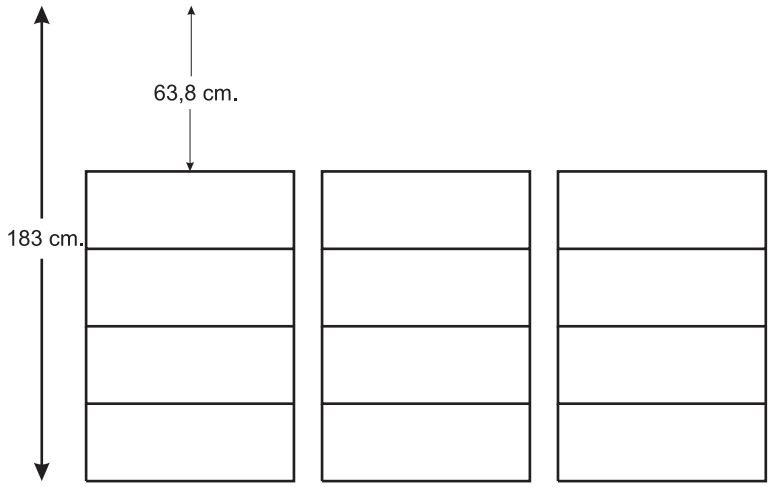


Figura 22: El matching iterado falla en la confección de transversales.

La situación es aún más grave: las rutinas de refinamiento revisarán estos bloques por pares. En este caso, la unica alternativa digna se ser analizada puede verse a ojo, y consiste en trasladar dos de las piezas de uno ellos al otro, dando como resultado un transversal de $178,8 \times 80$ cm. y un pequeño bloque suelto de $59,6 \times 80$ cm. Sin embargo, los largos de estos dos nuevos bloques suman $1,6$ mts.: exactamente lo mismo que los originales y, por lo tanto, la permutación no será realizada, pues no representa ninguna ganancia *a lo largo*.

En otras palabras: el método tal cual lo plantearon Fristch y Vornberger asume que al cabo de la segunda etapa los metarectángulos a considerar ocupan en su gran mayoría todo el ancho de la placa y que sólo resta, por lo tanto, realizar permutaciones que mejoren el largo. Esto puede ser cierto en el caso del vidrio pero, dadas las dimensiones de la placa, resulta falso en la mayor parte de los ejemplos sobre madera.

Para solucionar este inconveniente, implementamos nuevas rutinas de refinamiento, que actúan sobre los metarectángulos arrojados por la segunda etapa del algoritmo: tomando los bloques que aún no son transversales, se estudian, de manera similar a cómo lo hacían los refinados anteriores, posibles permutaciones de subárboles que den como resultado algún nuevo transversal. Es importante remarcar que estas rutinas pasan por alto el *largo* de los bloques, que será luego mejorado por los procedimientos ya expuestos, limitándose a actuar a lo ancho. Por otra parte, el objetivo no es simplemente mejorar la ocupación de la placa a lo ancho sino específicamente la formación de transversales.

Estas nuevas rutinas son también cuatro y se corresponden exactamente con las utilizadas para el mejoramiento de los transversales. Aquí exponemos sólo la cuarta, que omitimos en la sección anterior, por haber sido concebida inicialmente para esta fase particular del algoritmo.

En esta etapa los metarectángulos son grupos de pocas piezas, de modo que en la mayor parte de los casos no tendrá sentido un descenso de dos niveles. En cambio, resultan mucho más efectivos los refinamientos desbalanceados de bajo nivel. Por este motivo, la última rutina de refinado a lo ancho, esquematizada en la figura 23, actúa vinculando un metarectángulo entero con uno de los hijos de otro metarectángulo. El segundo hijo queda suelto, pasando a ser considerado como un nuevo metarectángulo activo en la siguiente iteración.

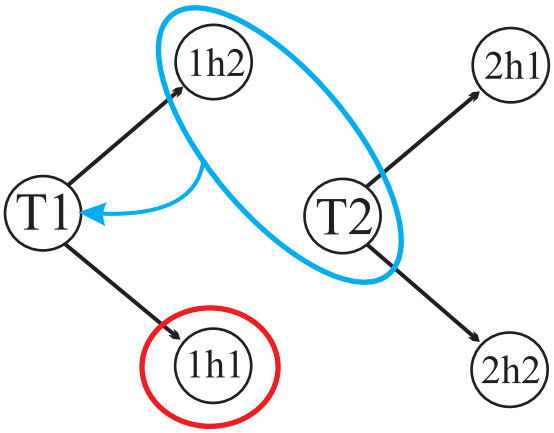


Figura 23: Refinado desbalanceado a un nivel.

Cada iteración finaliza con la realización de una permutación, que da como resultado un transversal y un bloque pequeño que queda suelto. Para la iteración siguiente, se descartan los transversales ya formados y se toman en consideración sólo los bloques que siguen dejando espacio a lo ancho. Este proceso se da por concluído cuando no pueden realizarse más permutaciones.

Ahora bien: al cabo de este procedimiento se tienen, por un lado algunos nuevos transversales y, por el otro, como subproducto, una serie de bloques pequeños que son las partes sobrantes de los metarectángulos permutados. En principio, entonces, nos encontramos en una situación similar a la anterior: si bien hemos formado nuevos transversales, sigue habiendo piezas que dejan un amplio sobrante a lo ancho y que la tercera etapa del algoritmo (tal cual lo plantearon Fristch y Vornberger), tomaría por transversales. La diferencia sustancial reside en el hecho de que estos bloques sobrantes son más pequeños que los bloques que ocasionaron el problema originalmente y son, por lo tanto, susceptibles de ser apareados a través de una nueva secuencia de matching iterado. Realizamos, entonces, un nuevo matching, que formará con los metarectángulos sobrantes bloques tan grandes como sea posible.

En la figura 24 se muestra el resultado de todo este proceso sobre el ejemplo mostrado anteriormente: en la fase de refinación, al analizar las permutaciones posibles entre el primer y el tercer bloque, la rutina de refinado desparejo en un nivel realiza el pasaje de una sub-bloque del tercero al primero, formando un transversal y dejando un sobrante formado por dos piezas. Luego, en la fase de matching, este sobrante se agrega al segundo bloque, formando un nuevo transversal. De este modo, lo que antes ocupaba 2, 4 mts. a lo largo (casi una placa entera) al costo de desperdiciar una gran superficie a lo ancho, pasa a ser dispuesto en sólo dos bloques, que aprovechan todo el ancho de la placa y dejan un metro libre a lo largo.

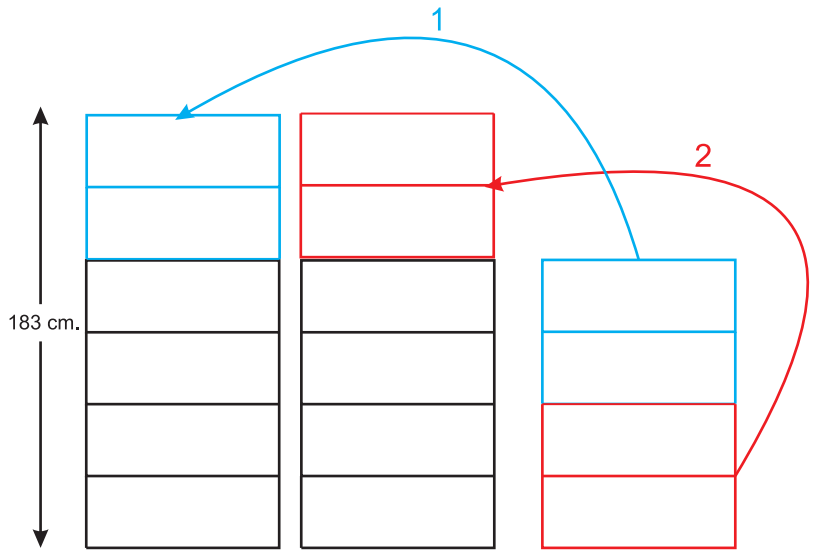


Figura 24: El refinado intermedio resuelve el problema.

Con estas modificaciones, el esquema de nuestra variante del algoritmo es:

1. **Primera Etapa:** Matching iterado para formar metarectángulos universales.
2. **Segunda Etapa:** Formación de transversales.
   - *Fase 1:* Matching iterado para formar transversales a partir de los universales. Aquí el peso de las ramas del grafo es disminuído en un 10 % en el caso en que el apareo forme un transversal e incrementado en igual medida en caso contrario.
   - *Fase 2:* Refinamiento iterado a lo ancho sobre los metarectángulos no transversales, para conseguir nuevos transversales.
   - *Fase 3:* Nuevo matching iterado, sólo sobre los metarectángulos no transversales, para agruparlos tanto como sea posible.
3. **Tercera Etapa:** Refinamiento a lo largo de todos los metarectángulos obtenidos, considerándolos como transversales aunque no lo sean.

4. **Cuarta Etapa:** Resolución del problema de `BIN PACKING` para disponer los metarectángulos tomados como transvesales sobre las placas.

5.6. Bin Packing

Al finalizar la construcción de transversales, para completar el diseño de los diagramas de corte es necesario distribuirlos sobre las placas, de manera de ocupar el mínimo número posible de éstas, resolviendo así un problema de `BIN PACKING`.

Así como favorecían la formación natural de transversales a través del matching iterado, las dimensiones de las placas de vidrio facilitan también su posterior disposición a lo largo de dichas placas. Los transversales tendrán entre un tercio y un cuarto de la longitud total de la placa, ocupando así longitudes más o menos homogéneas, lo que hace que el *First Fit Decreasing Algorithm* –el método más sencillo para abordar el problema de `BIN PACKING`,– aparezca como una estrategia razonable.

Ya hemos señalado las dificultades que presenta la fabricación de transversales en placas con relaciones largo-ancho más o menos chicas. Las técnicas utilizadas para atender estos problemas redundan ahora en nuevos inconvenientes. Puesto que nos vimos obligados a relajar las restricciones que definen un transversal, al cabo de la tercera etapa del algoritmo nos encontraremos con bloques de largos muy diversos, que pueden llegar a cerca de la mitad de la longitud de la placa. Bajo estas condiciones, el *First Fit Decreasing* puede resultar una muy mala elección, dado que comenzará ubicando juntos los bloques más grandes, que dejarán un sobrante importante pero que no puede ser completado por ningún otro transversal. Así, después del uso efectivo de las rutinas de refinamiento, la etapa final puede acumular enormes desperdicios, incluso en casos muy sencillos.

Con el objetivo de garantizar una disposición de los transversales que aproveche tanto como sea posible el largo de las placas, implementamos tres variantes del *First Fit Decreasing* cuyos resultados son comparados, quedándonos con el mejor de ellos. Estas variantes son:

1. El *First Fit Decreasing Algorithm*, tal cual como lo plantean Fristch y Vornberger.

2. Se listan los transversales por largo, de mayor a menor y se apartan aquellos con largos menores a un tercio de la longitud de la placa. Estos son unidos de a pares, por orden: los dos mayores, los dos siguientes, etc. Los bloques resultantes se incorporan a la lista de transversales, sobre la que se corre el *First Fit Decreasing*.

3. De manera análoga a lo hecho antes, se apartan de la lista los transversales *cortos* y se los une, pero siguiente un criterio más homogeneizador: se uno el mayor con el menor, etc. Nuevamente, los bloques resultantes se agregan a la lista como nuevos transversales y se distribuyen por *First Fit Decreasing*.

El programa estudia las tres variantes, guardando la información acerca de cuántas placas requiere cada método. Estos datos son luego comparados: la solución queda determinada por el método más eficiente.

Hemos constatado en la práctica que el estudio conjunto de estas variantes realmente mejora el rendimiento del algoritmo. Algunos ejemplos en los que el *First Fit* fallaba son resueltos satisfactoriamente por alguna de las rutinas alternativas. En otros, en cambio, las longitudes de los transversales son casi todas menores al tercio de la longitud de la placa, lo que hace que las versiones modificadas generen bloques innecesariamente grandes, dando peores resultados que el *First Fit* solo.

5.7. Una variante simplificada

Dado que en el caso de la madera no hay nada que obligue a realizar los primeros cortes en sentido transversal, implementamos también una versión del algoritmo que omite el paso de la contrucción de transversales.

La expectativa era que sin imponer restricciones al largo y sin beneficiar, mediante premios en los pesos de las ramas, la fabricación de transversales por sobre otras configuraciones con menor desperdicio, los resultados se vieran mejorados. Se preservaron, igualmente, todas las rutinas de refinamiento. De este modo, el algoritmo alternativo quedaría estructurado del siguiente modo:

1. **Primera Etapa:** Matching iterado, con peso igual al desperdicio promedio de la unión. No se separan ni universales ni transversales. La etapa finaliza cuando no hay más uniones posibles.
2. **Segunda Etapa:** Refinado a lo ancho:
   - *Fase 1:* Iteración de las rutinas de refinamiento a lo ancho.
   - *Fase 2:* Nueva secuencia de matching iterado, análoga a la anterior.
3. **Tercera Etapa:** Refinado a lo largo.
4. **Cuarta Etapa:** Resolución del problema de `BIN PACKING` para disponer los bloques sobre las placas.

La efectividad de esta variante del método, que llamaremos *directa*, no fue tan buena como esperábamos. Si bien en algunos ejemplos su rendimiento es mucho mejor que el del algoritmo *por transversales*, en otros da como resultado bloques prácticamente cuadrados, que aprovechan todo el ancho de la placa, pero sólo unos dos tercios del largo. De este modo resulta imposible ubicar dos de ellos en una misma placa, con lo cual el desperdicio, que es muy pequeño a lo ancho, se incrementa enormemente a lo largo.

En la siguiente sección mostramos una tabla comparativa, que da cuenta de las condiciones que deben cumplirse para que los resultados del método directo, que bloque por bloque suelen ser mejores que los del algoritmo por transversales, puedan ser aprovechados para obtener patrones de corte finales con menor desperdicio.

6. Algunos resultados obtenidos

Los resultados obtenidos con el algoritmo por transversales (convenientemente modificado) fueron muy satisfactorios. Aquí presentamos un ejemplo real de un conjuntos de 56 piezas, dado por la tabla 6. Las longitudes se dan en mm.

Ancho	Largo	Cantidad
200	1000	2
298	1000	12
298	800	14
600	747	2
600	770	4
600	300	4
370	550	4
370	414	4
410	277	2
135	410	2
165	305	6

El pedido fue distribuido en tres placas de 1830×2600 mm., con veta, sin admitir rotaciones. El tiempo requerido fue de 0,2 segundos. El desperdicio de los transversales de cada placa es de $6\,\%$, $3,3\,\%$ y $4,4\,\%$ respectivamente.

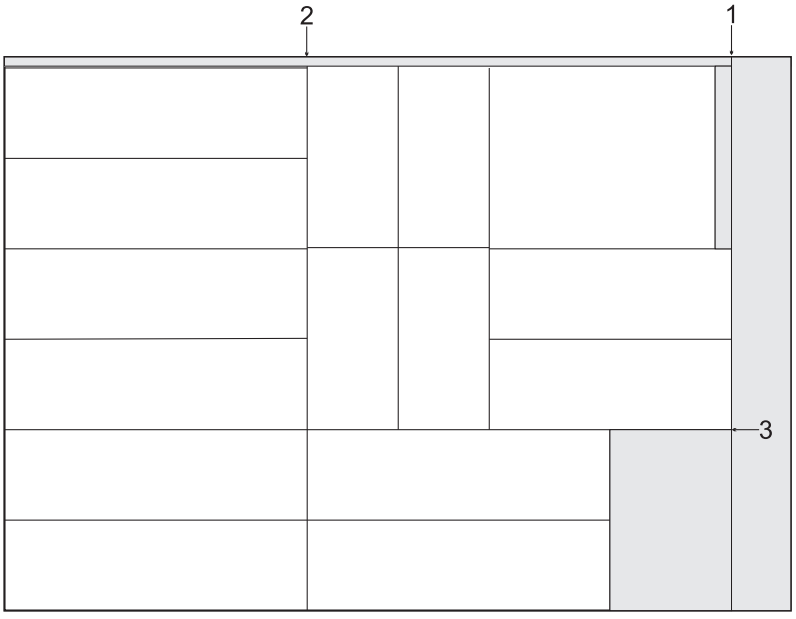


Figura 25: Primera Tabla. Ejemplo de 56 piezas.

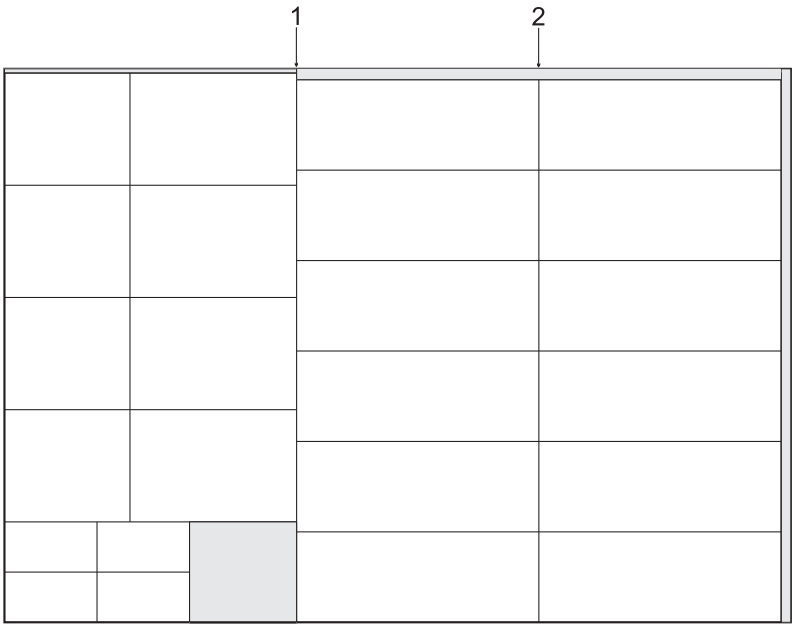


Figura 26: Segunda Tabla. Ejemplo de 56 piezas.

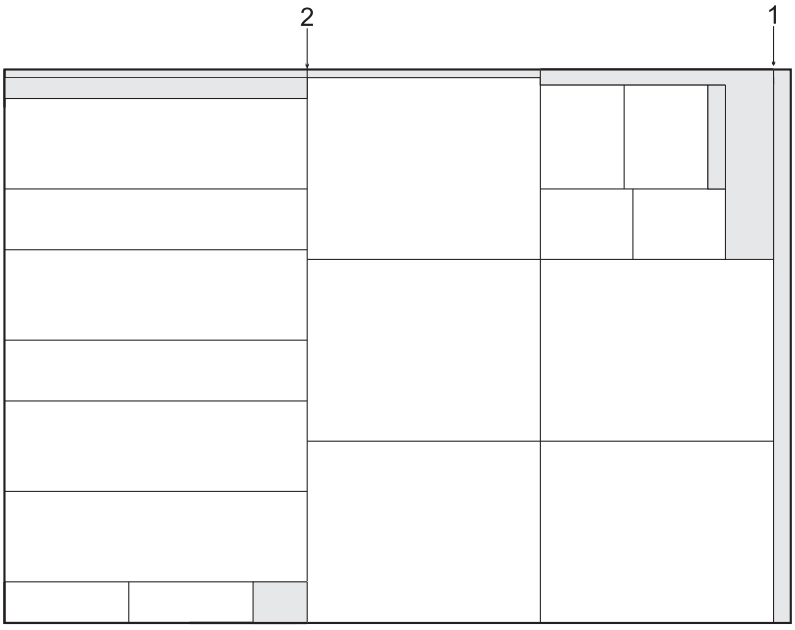


Figura 27: Tercera Tabla. Ejemplo de 56 piezas.

6.1. Método directo vs. Método por transversales

A modo de ejemplo, exponemos las soluciones a las que llegaron ambos métodos para un caso de 19 piezas, sin veta, en placa de 1830×2600.

Ancho	Largo	Cantidad
400	710	4
400	825	2
250	825	2
100	825	2
370	410	2
370	505	2
410	145	1
410	280	1
600	280	3

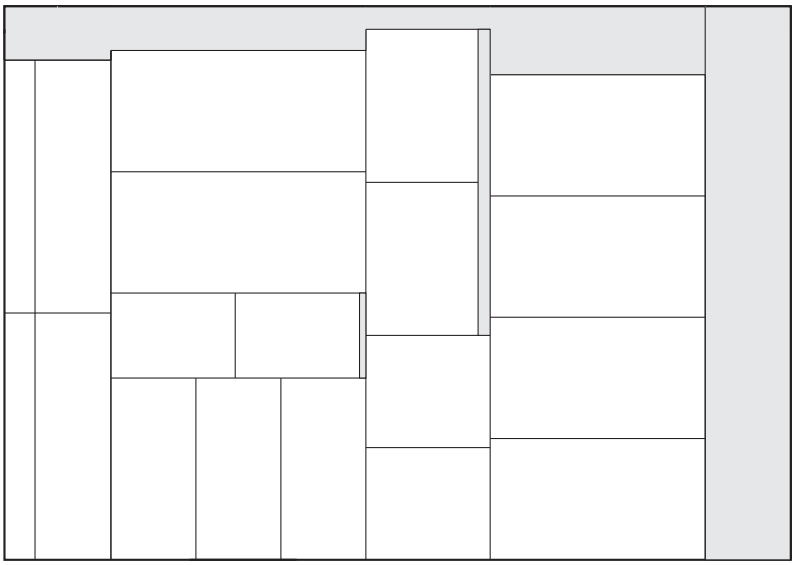


Figura 28: 19 piezas. Método Directo.

Si contabilizamos el desperdicio incluído en el área efectivamente utilizada de cada placa, tenemos un $12,4\,\%$ para el método por transversales y un $6,9\,\%$ para el directo.

Evidentemente, el métdo directo compacta mejor las piezas, acumulando menos desperdicio en el interior del metarectángulo final. Esto hace que resulte más práctico para instancias del problema en que las piezas no requieran más que en una única placa, puesto que en esos ejemplos dejará un mayor margen a lo largo, que puede ser luego reutilizado. Cuando hay muchas piezas, el rendimiento del método directo no siempre es tan bueno.

Debe tenerse en cuenta que nuestro principal objetivo es minimizar el número de placas utilizadas. A un mismo número de placas, preferiremos el patrón de corte que deje sobrantes enteros de mayor tamaño, más fáciles de reutilizar. El problema del método directo es que en muchos casos resulta en metarectángulos que, aún cuando contienen un desperdicio muy pequeño en su interior, ocupan más placas que la solución del método por transversales.

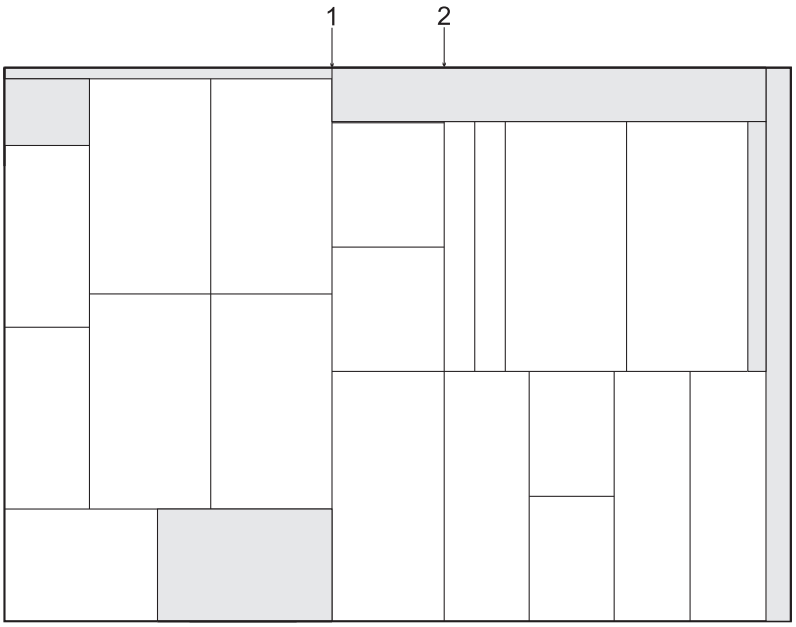


Figura 29: 19 piezas. Método por transversales.

Las tablas 6.1 y 6.1 evalúan el rendimiento de ambos métodos en varios ejemplos. Puede apreciarse que en la mayor parte de los casos, el método por transversales resulta más eficiente. En la columna *Desp. Int.* se muestra el desperdicio porcentual interno de los metarectángulos de la solución. En *Desp. Neto*, el desperdicio porcentual considerando la totalidad de las placas utilizadas.

Método Directo:

Cant.Piezas	Placas	Desp. Rel.	Desp. Neto
19	1	8,46 %	21,45 %
35	5	3,15 %	37,33 %
56	4	6,57 %	32,21 %
100	5	3,92 %	32,114 %
250	14	3,05 %	34,99 %

Método por Transversales:

Cant.Piezas	Placas	Desp. Rel.	Desp. Neto
19	1	9,1 %	21,25 %
35	4	4,24 %	21,66 %
56	3	5,19 %	9,61 %
100	4	4,70 %	15,13 %
250	10	2,17 %	8,98 %

Es importante remarcar que las áreas totales de las piezas nos garantizan, en todos los casos, que la solución arrojada por el método por transversales es óptima. Si queremos

mejorar estos resultados debemos buscar, dentro de cada placa, concentrar las piezas en un área menor, para conseguir así que los sobrantes sean más fáciles de reutilizar. El método directo cumple con este objetivo cuando se aplica en problemas chicos pero, como ya señalamos, pierde efectividad al ser utilizado sobre conjuntos de muchas piezas. Una posible continuación de este trabajo es, por lo tanto, el estudio de combinaciones de estos dos métodos que aprovechen las virtudes de ambos, contrarrestando sus defectos.

Referencias

[1] Cook, Stephen; **The Complexity of Theorem-Proving Procedures**; Proc. 3rd ACM Symp. on the Theory of Computing, ACM, 1971; pags. 151-158.

[2] Fritsch, Andreas y Vornberger, Oliver; **Cutting stock problem by iterated matching**; `http://www.inf.uos.de/papers_html/or_94`.

[3] Garey, M.R. y Johnson, D.S.; **Computers and Intractability** - A Guide to the Theory of NP Completeness; Freeman & Co.; 1979.

[4] Hertz, J.C.; **Recursive Computational Procedure for Two-dimensional Stock Cutting**; IBM J. Res. Develop.; Sept. 1972.

[5] Kernighan, B. y Ritchie, D.; **The C programming language**; Prentice Hall; 1988.

[6] Lawler, Eugene; **Combinatorial Optimization - Networks and Matroids**; Holt, Rinehart & Winston; 1976.

[7] Lovász, L.; **Computational Complexity** - Lecture Notes; 1994.

[8] Papadimitrou, C.H. y Steiglitz, K.; **Combinatorial Optimization - Algorithms and Complexity**; Dover; 1998.

[9] Ruiz, D.; **Programación C++**; MP Ediciones; 2004.

[10] Smart, J. y Hock, K.; **Cross Plataform GUI programming with** `wxWidgets`; Prentice Hall; 2005.

[11] Vicentini, Fabio; **Notas de Optimización**.

[12] Wong D.F. y Liu C.L.; **A New Algorithm for Floorplan Design**; IEEE, 1986.