Problema del empaquetamiento de texturas

Integrantes:
Alejandro García González **C411**
Carlos Mauricio Reyes Escudero **C411**

0. Introducción

La compresión de información es y siempre será un problema importante en computación. Si bien cuando el tamaño del almacenamiento o la velocidad de la red aumentan es posible manejar mayor cantidad de información, y más rápido, es difícil pensar que no aparecerá eventualmente un problema que pruebe los límites de la tecnología. En este trabajo nos centraremos en un caso particular de compresión: el empaquetamiento de texturas; esto es: dado un conjunto de imágenes, encontrar la mejor forma de compilarlas todas en una sola imagen. Un uso práctico de esto se encuentra en las tarjetas gráficas: convencionalmente las tarjetas gráficas tienen un número limitado (alrededor de 32) de los llamados "texture slots", que no son más que trozos de memoria de tamaño fijo destinados a guardar texturas; es entonces conveniente agrupar múltiples texturas en un solo texture slot, lo que además reduce la cantidad de cambios de estado interno de la GPU (cambiar de una textura a otra), y aumenta la localidad espacial.

1. Demostrando que el problema es NP-Duro

En esta sección demostraremos que un caso específico del problema es NP-Duro. Nos centraremos en cuando "mejor forma" se refiere a empacar imágenes en una sola imagen padre minimizando el área, y en particular cuando el ancho de la imagen padre es fijo. Formalizando:

Definición (Strip Packing problem): Dado un conjunto de rectángulos (con lados alineados a los ejes), y un ancho fijo W, hallar la mínima altura H para la cual se pueden empacar sin solapamiento todos los rectángulos del conjunto en un rectángulo de $W \times H$.

A continuación mencionamos la cadena de reducciones que seguiremos para demostrar que nuestro problema es difícil. La notación $Y \leq_p X$ significa "el problema Y se reduce polinomialmente al problema X".

3-SAT $\leq_p$ 3D-Matching $\leq_p$ 4-Partition $\leq_p$ Rectangle Packing $\leq_p$ Strip Packing

Definición (3D-Matching problem): Dado 3 conjuntos X, Y y Z disjuntos, cada uno de tamaño n, y un conjunto de triplas $T \subseteq X \times Y \times Z$, determinar si existe un $S \subseteq T$ tal que para todo $e \in X \cup Y \cup Z$, e aparece en exactamente una tripla $s \in S$.

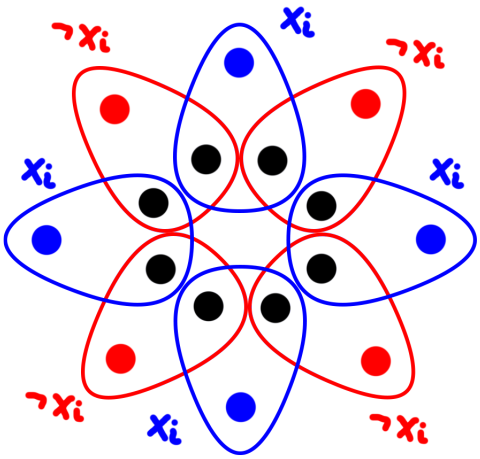
Informalmente: Dado una especie de "grafo" donde las "aristas" conectan a 3 nodos entre sí en lugar de solo a 2, y que es tripartito, hallar un "emparejamiento" (entriplamiento?) perfecto.

Teorema: 3D-Matching es NP-Completo.

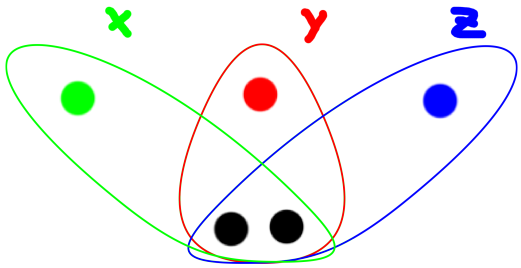
Demostración:

Es fácil ver que dado un conjunto de triplas, se puede comprobar en tiempo polinomial que dicho conjunto da un 3D-Matching completo válido. O sea, 3D-Matching $\in$ NP.

Ahora, para demostrar que es NP-Duro, reduzcamos 3-SAT a 3D-Matching. Dada una formula booleana en forma normal conjuntiva, hagamos lo siguiente:

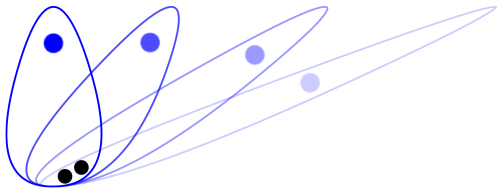


Por cada variable que aparece en la fórmula, construyamos un gadget como el que aparece en la figura. Este tiene el nombre de "gadget variable". Que 3 nodos estén encerrados dentro de un óvalo significa que están conectados entre sí (o sea, los óvalos son las "aristas" de esta especie de "grafo"). Pero la cantidad de picos del gadget será $2n_{x_i}$, donde n_{x_i} es la cantidad de veces que aparece la variable x_i (o su negación) en la fórmula. Este gadget tiene la peculiaridad de que los nodos que se muestran en negro no están conectados con nada fuera del gadget (se verá que los rojos y azules sí). Esto implica que para lograr un 3D-Matching perfecto, es necesario matchearlos utilizando los óvalos del gadget. Se comprueba entonces que para poder seleccionar a todos los nodos negros, hay solo dos posibilidades: o se eligen solo los óvalos azules o solo los rojos para estar en el matching (cualquier combinación de rojos y azules poseería solapamiento). Esto corresponde a elegir verdadero o falso para la variable x_i. En particular: si se escogen solo los óvalos rojos, o sea que se dejan libres los azules, esto correspondería a elegir verdadero para la variable; y viceversa si se escogen los azules. Esto es así porque dejar nodos libres permite a otros gadgets usarlos, como se verá a continuación.



Por cada cláusula $(x \vee y \vee z)$, hagamos la construcción que se muestra en la figura. Los nodos negros son locales (no están conectados a más nada fuera del gadget), pero los de colores son los exteriores de uno del gadget correspondiente a las variables x, y, z respectivamente (nos aseguramos anteriormente de que cada gadget variable tuviera suficientes picos para poder hacer esto sin solapamiento). Como en el gadget variable, si se quiere lograr un 3D-Matching perfecto, los nodos negros se tienen que matchear usando alguno de los óvalos del gadget (que por cierto, recibe el nombre de "gadget cláusula"). Esto corresponde a satisfacer esta cláusula usando el valor de una de sus variables. Entonces, satisfacer la fórmula sería equivalente a satisfacer todos los gadget cláusula.

Falta un detalle. Si la operación de dentro de una cláusula fuera un XOR en lugar de un OR, ya estaría la demostración terminada. Pero en una cláusula puede haber más de una variable satisfecha. Así que queda poner un "parche" para cuando esto pasa.



Creamos así un tercer y último gadget, que le llamaremos "gadget recolector de basura". Este consistirá en dos nodos locales (de nuevo los nodos negros de la figura), los que estarán conectados a todos los otros nodos pico de los gadgets variables. De esta forma, los nodos que "sobren" (o sea, que se hayan dejado libre en su gadget variable, y no se hayan usado para satisfacer algún gadget cláusula), se matchearán acá. La idea es usar $\sum n_{x_i} - \#clauses$ de estos gadgets, es decir, asumiendo que se hayan satisfecho todas las cláusulas, uno por cada nodo que se dejó sin matchear.

De esta forma ya tenemos una instancia (de tamaño polinomial) del problema de 3D-Matching que retorna verdadero ssi la fórmula que nos pasaron es satisfacible. ■

Definición (4-Partition problem): Dado un conjunto de n enteros $A = \{a_1, ..., a_n\}$, determinar si existe una partición del conjunto en $n/4$ subconjuntos, de 4 elementos cada uno, con la misma suma (que resulta tener que ser necesariamente $t = \frac{\sum A}{n/4}$).

Teorema: 4-Partition es NP-Completo.

Demostración:

Está claro que si se tiene una partición, comprobar que es válida es fácil: se suman los 4 números de cada subconjunto por separado, y se comprueba que todos den la misma suma. Luego, 4-Partition $\in$ NP.

Ahora, para demostrar que es NP-Duro, reduzcamos 3D-Matching a él. Entonces, dada una entrada al 3D-Matching, hagamos la siguiente construcción:

Vamos a llenar el conjunto que se quiere particionar para resolver el 3D-Matching con el 4-Partition. Procederemos a escribir los números en una especie de representación en base r, para un r lo suficientemente grande (para poder razonar mayormente dígito a dígito). En particular, usaremos $r = 100 \times 3n$, donde n es el tamaño de cada conjunto del 3D-Matching. En lo siguiente, la notación (a, b, c, d, e) representará al número $ar^4 + br^3 + cr^2 + dr + e$. En un momento usaremos "dígitos negativos", lo que justificaremos luego. Ahora, listemos por cada conjunto y tripla del 3D-Matching, qué números añadimos al conjunto a particionar, y cuál sería la suma objetivo (dictada por la estructura del conjunto, claro) y luego justifiquemos por qué resolver el 4-Partition en dicho conjunto es equivalente a resolver el 3D-Matching.

Por cada $x_i \in X \rightarrow (10, i, 0, 0, 1)$ y $(11, i, 0, 0, 1) \times (n_{x_i} - 1)$ copias
Por cada $y_j \in Y \rightarrow (10, 0, j, 0, 2)$ y $(11, 0, j, 0, 2) \times (n_{y_j} - 1)$ copias
Por cada $z_k \in Z \rightarrow (10, 0, 0, k, 4)$ y $(8, 0, 0, k, 4) \times (n_{z_k} - 1)$ copias
Por cada tripla $(x_i, y_j, z_k) \rightarrow (10, -i, -j, -k, 8)$
Suma objetivo (se demostrará en breves) $\rightarrow t = (40, 0, 0, 0, 15)$

Ahora a demostrar. Primero, demostremos que la suma es la suma objetivo es la que decimos allá arriba. La suma de cada subconjunto debe ser la suma total sobre un cuarto de la cantidad de elementos del conjunto. La cantidad de elementos del conjunto es $4 \times \#triplas$, ya que por cada tripla hay un elemento que la representa, y por cada número de la tripla hay un elemento que lo representa; o sea, cada tripla se cuenta otras 3 veces. La suma total del conjunto se halla sumando dígito a dígito. Esta termina siendo $(40r^4 + 15) \times \#triplas$. Dividiendo, queda demostrado que la suma es t.

Es importante justificar el uso de esos "dígitos negativos". Si bien en el contexto tienen sentido (no son más que coeficientes en una suma ponderada de potencias de r), en nuestro caso nos conviene que no interfieran con el dígito menos significativo, o sea, que independientemente de los valores de i, j y k, el valor del dígito menos significativo sigue siendo 8. Para esto, basta con que $10r^4 - ir^3 - jr^2 - kr \geq 0$ para los valores máximos de i, j y k, que son en los tres casos n (el tamaño de cada conjunto del 3D-Matching). Esto se logra sustituyendo el valor de r en función de n en esa expresión ($r = 3 \times 10^2 \times n$), sustituyendo i, j y k por su valor máximo n; luego de esto se reduce la expresión a una cuadrática con coeficiente principal positivo y ceros estrictamente menores a 1, lo que demuestra que la inecuación se cumple para todo valor de n, y por tanto para todo valor posible de i, j, k y r.

Ahora la pieza final de la demostración. Como la única forma de formar 15 sumando cuatro potencias de 2 es $8 + 4 + 2 + 1$, y el dígito menos significativo es

independiente, entonces para formar un cuarteto de números que sumen a t es necesario tomar uno que haya sido generado por el conjunto X, uno por el conjunto Y, uno por el Z y uno por una tripla. Al sumar un elemento generado por cada uno de X, Y y Z, los dígitos del medio quedan (i, j, k), por lo que hay que sumar el elemento generado por la tripla hay que cancelarlos, lo que implica que hay que sumar el elemento de la tripla (i, j, k). Por último, hay dos maneras de llegar a 40: sumando los 3 elementos únicos generados por cada conjunto (aquellos de dígito más significativo 10) junto a su tripla correspondiente, o sumando los elementos de los que se generan varios por nodo (dígito menos significativo 11, 11, 8, que cumplen $11 + 11 + 8 = 30$) junto a la tripla correspondiente. La primera alternativa corresponde a tomar la tripla (i, j, k) dentro del Matching, la segunda a no tomarla. Entonces, habrá 3D-Matching Perfecto ssi hay una forma de agrupar todos los elementos únicos generados por X, Y y Z, con una tripla correspondiente. ■

Definición (Rectangle Packing problem): Dado un conjunto de rectángulos (con lados alineados a los ejes), y un rectángulo más grande (también con lados alineados a los ejes) cuya área es la suma de las áreas de dichos rectángulos, determinar si es posible cubrir completamente el rectángulo grande con los otros sin solapamiento. No se permite rotar ningún rectángulo.

Teorema: Rectangle Packing es NP-Completo.

Demostración:

Si se tiene la posición de cada rectángulo pequeño, es fácil comprobar en tiempo polinomial que no haya solapamiento (basta con comprobar intersección entre todo par de rectángulos), y si no hay solapamiento, como la suma de sus áreas es igual a la del rectángulo grande, entonces dicho rectángulo está completamente cubierto. Luego, Rectangle Packing $\in$ NP.

Para demostrar que es NP-Duro, reduzcamos 4-Partition a él. Esta reducción es relativamente sencilla: convertimos cada elemento a_i del conjunto que se quiere particionar en un rectangulito de $a_i \times 1$, y ponemos que nuestro rectángulo base sea de $\frac{\sum A}{n/4} \times n/4$. Así cada fila del rectángulo grande representará un subconjunto, habiendo $n/4$ filas. ■

Teorema: Strip Packing es NP-Duro.

Demostración:

Reduzcamos Rectangle Packing a Strip Packing. Para resolver Rectangle Packing para un conjunto R de rectángulos y un rectángulo grande de $W \times H$, basta con usar un algoritmo para Strip Packing, pasándole R y W; si la salida de este algoritmo es H, existe un empaquetamiento perfecto (sin huecos), por lo que la respuesta a Rectangle Packing es True; si es distinta de H, tiene que ser estrictamente mayor que H, lo que significa que no es posible empaquetar los rectángulos de S sin solapamiento en un rectángulo de $W \times H$, por lo que la respuesta a Rectangle Packing es False. ■

2. Una solución exacta

Demostrado que el problema es NP-Duro, exploremos una alternativa para resolverlo exactamente que funciona bien en la práctica (o al menos, es de lo mejor que se puede hacer para resolver el problema exactamente). El algoritmo consistirá básicamente en reducir el problema a uno de programación en enteros, problema el cual es NP-Duro también, pero se conocen algoritmos prácticos para su resolución.

Asumiremos, sin pérdida de generalidad, que la entrada al algoritmo consiste en un conjunto de rectángulos (w_i, h_i), cada uno de los cuales está repetido d_i veces (o sea, hay exactamente d_i rectángulos de tamaño (w_i, h_i), están agrupados por tamaño), y el ancho fijo W del rectángulo grande. Lo primero que hará el algoritmo es calcular una cota mínima para H. En la práctica se utiliza a menudo el área total de los rectángulos sobre el ancho del rectángulo grande. Esto es: asumiendo que el empaquetamiento quede perfecto (sin huecos, sin espacio desperdiciado), ¿cuál sería H?. Formalmente, la cota es:

$$H \geq \left\lceil \frac{\sum d_i w_i h_i}{W} \right\rceil$$

Ahora, comenzando con este valor menor que H, y mientras no se haya hallado el H óptimo, se realiza un proceso para ver si el H actual es el óptimo. Describamos este proceso.

Por conveniencia, enumeremos linealmente los píxeles de la imagen grande, comenzando por uno en la esquina superior izquierda y terminando por $W \times H$. Ahora, por cada imagen pequeña i, haremos una tabla $C^i = c^i_{jp}$. Cada fila corresponderá a una posición válida de la imagen, es decir, a un pixel en el que poner la esquina superior izquierda de la imagen tal que dicha imagen no se pase de los límites de la imagen padre, y habrá una columna por cada pixel de la imagen padre, de forma tal que el elemento c^i_{jp} es 1 si para la imagen i, y la j-ésima posición válida de la esquina superior izquierda $(V[j])$, la imagen ocupa el pixel p, y en caso contrario 0.

Ahora la formulación lineal del problema. Llamaremos a las variables de decisión x_{ij}, y tendremos una por cada imagen i, y cada posición válida j de su esquina superior izquierda; así, $x_{ij} = 1$ si se decide que la imagen i se debe posicionar en su j-ésima posición válida $V[j]$, y 0 en otro caso. Entonces, sea P el conjunto de posiciones de la imagen grande, I el conjunto de imágenes, y F^i el conjunto de las filas de la tabla de la imagen i, las restricciones a satisfacer son las siguientes:

$$\sum_{i \in I} \sum_{j \in F^i} c^i_{jp} x_{ij} \leq 1 \quad \forall p \in P$$

$$\sum_{j \in F^i} x_{ij} = d_i \quad \forall i \in I$$

La primera restricción garantiza que por cada pixel de la imagen padre, haya a lo sumo una imagen ocupándolo. La restricción se lee: "por cada pixel p de la imagen

grande, pon un contador en cero, itera por todas las imágenes, y si para alguna posición válida de la esquina superior izquierda escogida de dicha imagen se usa el pixel p, suma uno al contador; para todos los casos el contador debe ser menor o igual a 1".

La segunda restricción es más sencilla, está ahí para que la cantidad de imágenes del mismo tamaño que se pongan sea la correcta. Resulta que tratar a las imágenes del mismo tamaño en batch ayuda al algoritmo a converger. En la misma línea, se pueden añadir otras restricciones para ayudar al algoritmo, en las que no entraremos.

La complejidad del algoritmo sería $O(\, H \cdot ((HW)^2 n + i(HWn))\,)$, donde $i(x)$ es la complejidad del algoritmo utilizado para resolver el problema de programación lineal, y H es la altura óptima. Si se quiere expresar la complejidad puramente en función de la entrada, se puede acotar superiormente H por la suma de las alturas de las imágenes. Si se es un poco más cuidadoso al construir la tabla de posiciones válidas, y se aprovecha la construida para la iteración anterior, es posible mejorar un poquito la complejidad.

3. Greedy

Ahora, planteemos una restricción al Strip Packing Problem: que todos los rectángulos son del mismo tamaño (w, h). En este caso para hacer el empaque óptimo es suficiente un algoritmo greedy sencillo: comenzando por la esquina superior izquierda, ir poniendo los rectángulos en fila, hasta que se agote el ancho; en ese momento, se continúa haciendo lo mismo desde la esquina superior izquierda del espacio aún disponible (debajo del primer rectángulo que se posicionó), y así se continúa, llenando la imagen de izquierda a derecha, de arriba a abajo, hasta que se acaban los rectángulos. Es fácil ver que la altura del rectángulo resultante de este algoritmo será:

$$H = \left\lceil \frac{n}{\left\lfloor \frac{W}{w} \right\rfloor} \right\rceil \cdot h$$

Quedaría justificar la optimalidad de esta respuesta. Tomemos un empaquetamiento óptimo cualquiera. Si nos fijamos en una fila de píxeles cualquiera, se cumple que la cantidad de rectángulos que usan esa fila será menor o igual a $\left\lfloor \frac{W}{w} \right\rfloor$, lo que implica que la columna de píxeles que colisiona con más rectángulos tiene que colisionar con por lo menos $\frac{n}{\left\lfloor \frac{W}{w} \right\rfloor}$, lo que a su vez significa que la altura es mayor o igual a la H que halla el algoritmo greedy, con lo que queda probada la optimalidad.

4. Algoritmos de Aproximación

Dado que el problema del empaquetamiento de tira es NP-Duro, en esta sección exploraremos algoritmos de aproximación. Estos algoritmos, si bien no garantizan la solución óptima, ofrecen soluciones cercanas a la óptima en tiempo polinomial. Nos centraremos en algoritmos 'orientados a niveles' (level-oriented), que organizan los rectángulos en niveles horizontales dentro del contenedor.

4.1 Next-Fit Decreasing Height (NFDH)

NFDH es un algoritmo sencillo e intuitivo. Primero, se ordenan los rectángulos por altura, de mayor a menor. Luego, se comienza a 'empacar' los rectángulos en un 'nivel' inicial que es la base del rectángulo grande (de ancho W).
1. Creación de Niveles: El primer rectángulo (el más alto) define la altura del primer nivel.
2. Empaquetamiento en el Nivel Actual: Se van colocando los siguientes rectángulos, uno al lado del otro, hasta que el siguiente rectángulo no quepa.
3. Nuevo Nivel: Cuando un rectángulo no cabe en el nivel actual, se crea un nuevo nivel 'encima' del nivel anterior.
4. Repetición: Se repiten los pasos 2 y 3 hasta que se hayan empaquetado todos los rectángulos.

NFDH tiene las siguientes cotas de aproximación [Coffman et al., 1980]:
1. Cota Asintótica: $NFDH(L) \leq 2 \cdot OPT(L) + h_{max}$, donde h_{max} es la altura del rectángulo más alto.
2. Cota Absoluta: $NFDH(L) \leq 3 \cdot OPT(L)$.
La complejidad temporal es de $O(n \cdot \log n)$ pues solo ordena los rectángulos y luego los coloca uno a uno.

4.2 First-Fit Decreasing Height (FFDH)

FFDH es una mejora sobre NFDH. También ordena los rectángulos por altura decreciente, pero en lugar de crear un nuevo nivel cada vez que un rectángulo no cabe, intenta colocarlo en cualquier nivel existente donde quepa.
1. Ordenamiento: Igual que NFDH, se ordenan los rectángulos por altura decreciente.
2. Empaquetamiento en el Primer Nivel Disponible: Para cada rectángulo, se recorren los niveles existentes, y si el rectángulo cabe en un nivel, se coloca en ese nivel.
3. Nuevo Nivel (solo si es necesario): Si el rectángulo no cabe en ningún nivel existente, se crea un nuevo nivel encima del nivel más alto actual, y se coloca el rectángulo allí.
4. Repetición: Se repite el paso 2 y 3 hasta empaquetar todos los rectángulos.

FFDH tiene las siguientes cotas de aproximación [Coffman et al., 1980]:
1. Cota Asintótica: $FFDH(L) \leq 1.7 \cdot OPT(L) + h_{max}$.
1. Cota Absoluta: $FFDH(L) \leq 2.7 \cdot OPT(L)$.
La complejidad temporal es de $O(n^2)$, pues debe revisar todos los niveles, y puede haber un nivel por rectángulo.

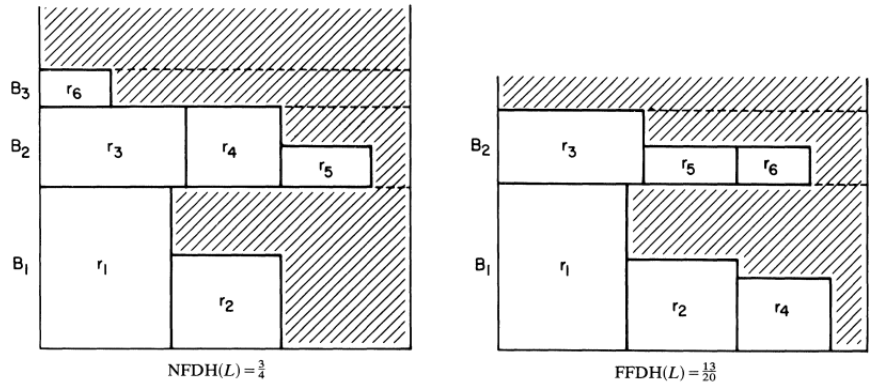


Figura 1: Comparación entre NFDH y FFDH.

4.3 Reverse-Fit (RF)

Reverse-Fit introduce una idea diferente: alternar la dirección de empaquetamiento. A continuación, se detallan los pasos de este algoritmo:
1. Preprocesamiento (Rectángulos Anchos): Se apilan todos los rectángulos de ancho mayor que $W/2$ uno encima del otro en la parte inferior del contenedor. Se define H_0 como la altura total de esta pila, que servirá como la base para el resto del empaquetamiento.
2. Ordenamiento: Se ordenan los rectángulos restantes (los de ancho menor o igual a $W/2$) en orden decreciente de altura.
3. Primer Nivel (Izquierda a Derecha): Se empaquetan los rectángulos, comenzando desde la base H_0, de izquierda a derecha. Se llena este primer nivel hasta que no quepan más rectángulos.
4. Segundo Nivel (Derecha a Izquierda - Reverse Level): Se crea un nuevo nivel encima del primer nivel. Este nivel se llena en dirección opuesta, de derecha a izquierda y alineados al techo dado por la línea en $H_0 + h_{max} + d_1$ donde h_{max} es la altura del rectángulo más alto incluido en el primer nivel y d_1 es el primer rectángulo a incluir en este nivel. Se empaquetan rectángulos hasta que este nivel tenga una ocupación de al menos $W/2$ o ya no queden más rectángulos.
5. Ajuste del Segundo Nivel: Se "bajan" los rectángulos del segundo nivel hasta que toquen los rectángulos del primer nivel. Se define H_1 como la altura del segundo nivel después de este ajuste.
6. Tercer Nivel (Si es Necesario): Si el ancho del segundo nivel no es al menos $W/2$, se realiza un proceso más complejo para definir un tercer nivel.
7. Niveles Subsiguientes: A partir del tercer nivel, se continua con un enfoque de "First Fit" modificado, creando nuevos niveles según sea necesario y llenándolos de izquierda a derecha.

Reverse-Fit tiene la cota absoluta de aproximación $RF(L) \leq 2 \cdot OPT(L)$.

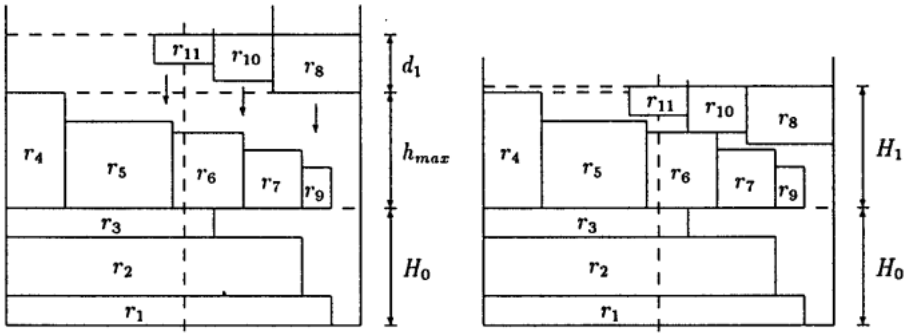


Figura 2: Primeros dos niveles de RF y caída libre del bloque pegado.

4.4 Dificultad de Aproximación

La búsqueda de mejores algoritmos de aproximación para el Strip Packing no es solo un ejercicio teórico; existen límites fundamentales a qué tan bien podemos aproximar este problema. Específicamente, cuando se permite que las dimensiones de los objetos crezcan exponencialmente en relación al tamaño total de la entrada, se ha demostrado que no se puede aproximar el problema con un factor mejor a 3/2, a menos que P = NP. Esto se puede demostrar mediante una reducción directa desde el problema de la Partición.
Sea $C = \{a_1, \ldots, a_n\}$ un conjunto de enteros positivos cuya suma es $2X$. Construimos una instancia de Strip Packing definiendo el ancho de la tira como $W = X$ y, para cada $a_i \in C$, un rectángulo de dimensiones $a_i \times 1$. Si existe una partición de C en dos subconjuntos de suma X cada uno (solución positiva del problema PARTITION), se pueden empacar los rectángulos en dos filas de altura 1, obteniéndose un empaquetado de altura óptima $OPT(L) = 2$. En caso contrario, cualquier partición forzará que al menos una de las filas exceda el ancho X, haciendo necesario el uso de una tercera fila y forzando $OPT(L) \geq 3$. Por lo tanto, si existiera un algoritmo A de aproximación polinomial tal que, siendo $k < \frac{3}{2}$, se cumpla que $A(L) \leq k \cdot OPT(L)$, al aplicarlo a esta instancia se obtendría $A(L) < 3$ en el caso positivo y $A(L) \geq 3$ en el caso negativo, lo que permitiría decidir en tiempo polinomial si la instancia de PARTITION tiene solución, contradiciendo la NP-completitud del problema. El mejor algoritmo de aproximación conocido hasta la fecha tiene una cota de 5/3 por [Harren et al., 2014].
En [Henning et al., 2018] se demuestra que el límite inferior para un algoritmo que se ejecute en tiempo pseudopolinomial es de 5/4, el cual ya existe(ver [Jansen and Rau, 2019]).

References

[Coffman et al., 1980] Coffman, Jr., E. G., Garey, M. R., Johnson, D. S., and Tarjan, R. E. (1980). Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4):808–826.

[Harren et al., 2014] Harren, R., Jansen, K., Prädel, L., and van Stee, R. (2014). A (5/3+)-approximation for strip packing. *Computational Geometry*, 47(2, Part B):248–267. The 12th International Symposium on Algorithms and Data Structures (WADS2011).

[Henning et al., 2018] Henning, S., Jansen, K., Rau, M., and Schmarje, L. (2018). Complexity and inapproximability results for parallel task scheduling and strip packing. In Fomin, F. V. and Podolskii, V. V., editors, *Computer Science – Theory and Applications*, pages 169–180, Cham. Springer International Publishing.

[Jansen and Rau, 2019] Jansen, K. and Rau, M. (2019). Closing the Gap for Pseudo-Polynomial Strip Packing. In Bender, M. A., Svensson, O., and Herman, G., editors, *27th Annual European Symposium on Algorithms (ESA 2019)*, volume 144 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 62:1–62:14, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.