PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

Problema de Carga de Contenedores (Container Loading Problem)

KEITEL GUERRERO FUENTES
EDUARDO NÚÑEZ PARRA

MAYO, 2016

PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

Problema de Carga de Contenedores (Container Loading Problem)

KEITEL GUERRERO FUENTES
EDUARDO NÚÑEZ PARRA

Ignacio Araya Zamorano
Profesor Guía

José Rubio León
Profesor Co-referente

MAYO, 2016

AGRADECIMIENTOS

Les agradezco a mis padres y hermanas que siempre estuvieron conmigo en todo este proceso, a mi polola e hija Isabella y a todos quienes tuvieron palabras de apoyo a lo largo de mi carrera. Además al profesor Ignacio Araya quien nos apoyó y aconsejo en todo el proceso del proyecto.

Eduardo Núñez P.

A mi familia que siempre acompaña, incentiva y resguarda cada uno de los pasos que doy y daré. A la escuela de informática que me dotó de las herramientas y vivencias que acompañarán mis nuevos días. A mi fiel compañero de estudios nocturnos RocoRocus, a Yeyo que sigue iluminándonos desde algún lugar y a cada persona que conocí durante de este placentero y vertiginoso proceso. A todos y cada uno de ustedes… ¡Gracias infinitas!

Keitel Guerrero

Resumen

En la actualidad, existe una serie de problemas que surgen al interior de industrias de distribución y transporte. Uno de ellos es el *problema de carga de un contenedor único* (CLP), que consiste básicamente en llenar un único contenedor con un conjunto de cajas dadas, ello con el fin de maximizar la utilización del espacio dentro del contenedor.

Este trabajo estudiará en profundidad los componentes de uno de los algoritmos más exitosos para resolver este problema; así como también, se diseñarán nuevas heurísticas y métodos para mejorar el desempeño del algoritmo en cuestión. El desempeño de las técnicas propuestas se evaluará utilizando una serie de 1600 casos de prueba, que serán comparados con los resultados obtenidos por diferentes algoritmos del estado del arte.

Palabras clave: CLP, Heurísticas, Maximizar, Algoritmo.

Abstract

Currently, there are a series of problems that arise within distribution and transportation industries. One is the problem of loading a single container, (CLP), which basically consists of filling a single container with a given set of boxes, the objective is to maximize the use of space within the container.

In this paper the components of one of the most successful algorithms for solving the problem will be studied in depth; then, new heuristics and methods will be designed to improve the performance of the studied algorithm. The performance of the proposed techniques will be evaluated using a series of 1600 benchmark instances, and they will be compared with the results obtained by different algorithms of previous works.

Keywords: CLP, Heuristics, Maximize, Algorithm.

Índice

Índice de Figuras

Lista de Tablas

1 Introducción

El problema de carga de un contenedor único (CLP) es un problema de embalaje tridimensional, donde un contenedor debe ser llenado por un conjunto de cajas con el objetivo de maximizar la utilización del espacio dentro de éste.

La mayoría de los acercamientos para resolver este problema han sido creados basados en algoritmos constructivos (construyen bloques de cajas para la búsqueda de la solución). En este sentido, los algoritmos exactos que han sido propuestos son muy pocos, de los que destacan Fekete y Schepers [4] que presenta un marco general para múltiples problemas de embalaje dimensionales, y Martello [5] quien desarrolló un método exacto llamado branch and bound para CLP.

Hoy en día, donde el tiempo que se demora el algoritmo en encontrar la solución es crucial, las heurísticas y meta-heurísticas son la única alternativa viable para encontrar buenas soluciones. Dicho esto, este trabajo contempla el diseño e implementación de heurísticas que buscarán la mejor aproximación a la solución óptima del problema. Para esto, se tomará como base el algoritmo Beam-Search-Greedy (BSG-CLP) propuesto por Araya y Riff (2014) [3], que es el que mejor resultados presenta en la búsqueda de la solución hasta el momento.

En general, CLP se puede considerar como una generalización tridimensional del problema de la mochila, donde se pueden presentar diversos casos; tener muchos tipos de cajas (fuertemente heterogéneo), unos pocos tipos de cajas (débilmente heterogéneos), o un solo tipo de caja (homogéneo). Para efectos de este trabajo, la solución que se diseñe será única para todos los escenarios. Cabe mencionar que, para una mayor estructura, la heurística planteada se trabajará sobre el postulado de Wenbin Zhu (2012) [1], quien asegura que "gran parte de las estrategias utilizadas para resolver el CLP pueden ser descritas en 6 elementos claves". Cabe mencionar que para entender el algoritmo y las propuestas de manera más clara se usará la descomposición propuesta por Wenbin Zhu (2012) [1], quién divide los algoritmos de resolución en 6 componentes clave.

Finalmente, por medio de diferentes instancias (casos de prueba), se comparará la efectividad de la heurística planteada, con respecto a la variación del BSG-CLP, propuesta por Araya y Riff (2014) [3], que es el trabajo que mayor optimiza el espacio del contenedor, hasta ahora.

2 Definición de objetivos

A continuación, se presentan los objetivos propuestos para el proyecto, los cuales deberán ser considerados para el desarrollo y análisis del algoritmo.

2.1 Objetivo general

Diseñar heurísticas para mejorar el desempeño del algoritmo BSG-CLP orientado a la resolución del problema de carga de un único contenedor.

2.2 Objetivos específicos del algoritmo

Dentro de los objetivos específicos del proyecto, se encuentran:

- Comprender el problema de carga de contenedores y los métodos de resolución existentes.
- Diseñar e implementar heurísticas para el algoritmo BSG-CLP
- Realizar experimentos para evaluar el desempeño de las heurísticas propuestas y compararlas con el estado del arte

3 Estado del arte

El SCLP (problema de carga de un único contenedor) es una de las muchas variantes de problemas de corte y embalaje en la investigación de las operaciones; para el lector interesado, hacemos referencia al excelente libro de Dyckhoff y Finke [8], donde se ofrece un estudio detallado sobre el tema.

La utilización del volumen es el objetivo primario y más frecuente para muchos problemas de carga de contenedores del mundo real; un embalaje de gran densidad puede resultar en el menos uso de vehículos con contenedores, que también tiene un efecto biológicamente beneficioso para las emisiones de carbono. Otras restricciones, como la estabilidad de carga, cargas multi-drop, distribución del peso y facilidad de recuperación también se considera a menudo en aplicaciones específicas [7, 2, 18]. Una generalización útil del SCLP es considerar múltiples recipientes de diferentes tipos y costes, donde el objetivo es cargar todas las cajas de manera que el total de coste de los contenedores se reduce al mínimo; esto puede ser considerado una generalización tridimensional del problema de embalaje que ha sido examinada por Che y otros. [21].

Es posible clasificar los algoritmos existentes para SCLP en tres clases. Método Constructivo, que genera soluciones mediante la carga de cajas en repetidas ocasiones en el recipiente, hasta que ya no se puedan colocar más cajas. Dividir y conquistar, este método divide el recipiente en subcontenedores, para luego recursivamente resolver el problema menor resultante antes de recombinarlo en una solución completa; ejemplos incluyen Chien y Wu [6], Lins et al. [14]. Finalmente, métodos de búsqueda local, los cuales comienzan con una solución existente, luego aplican repetidamente operadores locales para generar nuevas soluciones; ejemplos incluyen Gehring y Bortfeldt [12], Parreño et al. [16].

Métodos basados en construcción de bloques son el foco de este estudio. Un bloque es un subconjunto de cajas que se colocan de forma compacta dentro de su delimitador mínimo cuboidal, los cuales se crean luego de ver el espacio residual del contenedor. Cada paso en la construcción de la solución implica la colocación de un bloque en un espacio libre del contenedor. Esto se repite hasta que no hayan más bloques que quepan en el contenedor. La ventaja de estos métodos es que van creando varias soluciones y estas soluciones se van mejorando con el transcurso del tiempo hasta llegar a un óptimo local.

Los métodos de construcción de bloques incluyen los algoritmos desarrollados por Bortfeldt et al. [11], Eley [9], Lim et al. [14], Mack et al. [15]. Los métodos recientes más exitosos para SCLP en los casos de prueba estándar han sido enfoques basados en construcción de bloques. Parreño et al. [16] introdujo un algoritmo de espacios maximales que utiliza una búsqueda GRASP en dos fases. El mejor algoritmo estudiado durante este trabajo fue el BSG-CLP propuesto por Araya y Riff [3]. El algoritmo utiliza un árbol de búsqueda incompleto (beam search) para construir las soluciones del problema. Las soluciones parciales son evaluadas usando un algoritmo voraz.

También se han aplicado enfoques de construcción de muros [10, 18], donde el contenedor es llenado por capas verticales (llamadas muros o paredes), y acercamientos de construcción de capas [2, 19], donde el contenedor se llena desde la parte inferior usando capas

horizontales. Tanto los muros como las capas horizontales pueden verse como tipos especiales de bloques, por lo que estos enfoques también pueden ser considerados como enfoques basados en construcción de bloques. Todas estas técnicas generan bloques sólo cuando la combinación de cajas contenidas dentro se considera conveniente por alguna medida de calidad (por ejemplo, si el volumen total de las cajas cubre como mínimo el 98% de todo el volumen del bloque).

4 Modelo matemático

Para contextualizar de mejor manera este problema, se hará una representación matemática de este, donde a través del lenguaje matemático se definirán los componentes, restricciones y otras observaciones que engloba esta problemática. Dicho esto, el problema de carga de contenedores puede enunciarse de la siguiente manera:

Dado un contenedor T de dimensiones L, W, H y un conjunto de n cajas C: $\{c_1; c_2; ...; c_n\}$, se desea acomodar un número determinado de cajas con el fin de maximizar el volumen total de estas, cargadas dentro del contenedor T. Cabe mencionar que las cajas no pueden superponerse unas con otras y las caras de las cajas estarán paralelas a las del contenedor. Junto con esto se definirán una serie de elementos y consideraciones que ayudarán a ejemplificar y comprender ésta problemática.

4.1 Parámetros

- $l(c_i)$: largo de una caja i en su estado original (antes de ser cargada en el contenedor).
- $w(c_i)$**:** ancho de la caja en su estado original.
- $h(c_i)$: altura de la caja en su estado original.
- $V(c_i)$: volumen de una caja i. Se puede obtener multiplicando sus dimensiones, i.e.,

$$V(c_i) = l(c_i) * w(c_i) * h(c_i)$$

4.2 Variables

- l_i: Tamaño de la caja en la dirección del eje x una vez colocada dentro del contenedor.
- w_i: Tamaño de la caja en la dirección del eje y una vez colocada dentro del contenedor.
- h_i: Tamaño de la caja en la dirección del eje z una vez colocada dentro del contenedor.
- x_i: Corresponde a la posición del vértice más cercano al origen en el eje x de la caja i.
- y_i: Corresponde a la posición del vértice más cercano al origen en el eje y de la caja i.
- z_i: Corresponde a la posición del vértice más cercano al origen en el eje z de una caja i.

Consideración: Las dimensiones (l_i, w_i, h_i) no coinciden necesariamente con las dimensiones originales de la caja, ya que ésta puede ser orientada de manera distinta a la original.

4.3 Función objetivo

Formal y matemáticamente este problema se puede expresar de la siguiente manera:

Max $\sum_{i=0}^{n} p_i * V(c_i)$

Donde $V(c_i) = l(c_i) * w(c_i) * h(c_i)$ corresponde al volumen de la caja C_i y P_i es una variable booleana que indica si la caja se coloca, o no, en el contenedor.

4.4 Restricciones

En este punto se detallarán diferentes restricciones que se deben tener en cuenta para evitar problemas de superposición de cajas, cajas fuera de los márgenes del contenedor u orientación errónea de cajas. Cabe mencionar que se obviaron algunas restricciones adicionales que se abordarán más adelante.

4.4.1 Orientación

Una caja en particular c_i está representada por sus dimensiones $l(c_i)$, $w(c_i)$y $h(c_i)$ que son, respectivamente, la longitud, ancho y altura en su posición original. Cabe mencionar que, para efectos prácticos, esta caja se podrá colocar en el contenedor en la posición más conveniente, con la salvedad de mantener sus caras paralelas a las caras de contenedores. Las posibles orientaciones son representadas por la siguiente restricción:

$(l_i, w_i, h_i)=(l(c_i),w(c_i),h(c_i))\vee$

$(l_i, w_i, h_i)= (l(c_i),h(c_i),w(c_i))\vee$

$(l_i, w_i, h_i)= (w(c_i),l(c_i),h(c_i))\vee$

$(l_i, w_i, h_i)= (w(c_i),h(c_i),l(c_i))\vee$

$(l_i, w_i, h_i)= (h(c_i),l(c_i),w(c_i))\vee$

$(l_i, w_i, h_i)= (h(c_i),w(c_i),l(c_i))$

Para todo $i = \{1 \dots n\}$

4.4.2 Traslapamiento

Esta restricción indica que no pueden existir pares de cajas compartiendo un mismo volumen.

Si $p_i = p_j$, entonces:

- $(x_i + l_i \leq x_j) \vee (y_i + w_i \leq y_j) \vee (z_i + h_i \leq z_j) \; \forall j = 1..n, \; j \neq i$

Para no infringir esta restricción se deben dar, una de 3 situaciones. Primero, la suma del punto de origen de la primera caja con el largo de la misma debe ser menor o igual al punto de origen de la caja j, todas estas considerando su valor en el eje x. Los otros 2 casos funcionan de la misma manera pero con los otros ejes (y;z), con sus respectiva métrica (altura y profundidad).

4.4.2.1 Ejemplo de traslapamiento

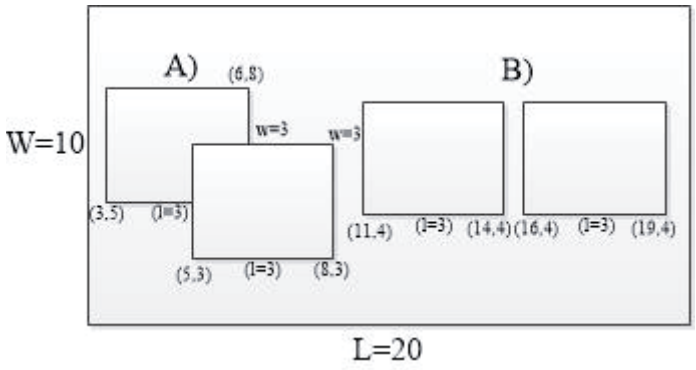

Figura 4.1 Restricción eje Xi

En la figura 4.1 se muestra cómo actúa la restricción del traslapamiento en el eje Xi en 2 dimensiones. Como se puede ver en la figura A se ve claramente como las cajas están sobrepuestas. Si evaluamos la restricción esta no se cumple:

A) $x_i = 3, x_j = 5, y_i = 3, y_j = 3, l_i = 3, w_i = 3, l_j = 3, w_j = 3$

$3+3 \leq 5 \vee 5+3 \leq 5$

$6 \leq 5 \vee 8 \leq 5 \Leftrightarrow \mathbf{F}$

El cálculo de esta restricción está dado por 2 escenarios, si fallan ambos, se aplica la restricción. El primer escenario calcula la suma de las coordenadas en el eje Xi del primer bloque en donde la suma de su origen más el largo no pueden superar el origen del bloque adyacente, donde claramente se observa que la restricción no se cumple $3+3 \leq 5$. El segundo escenario funciona de manera similar pero con los valores del eje Wi de los bloques, lo que arroja otro incumplimiento de la restricción $5+3 \leq 5$. Por todo lo anteriormente mencionado es que el programa actualmente no permite el posicionamiento del bloque que invade al antecesor, dado que real y matemáticamente no es posible.

Por otro lado, las cajas en la figura B cumplen la restricción, ya que sumando el punto de origen con su medida correspondiente, no infringen la restricción: 11+3 ≤ 16 o 4+3 ≤ 7.

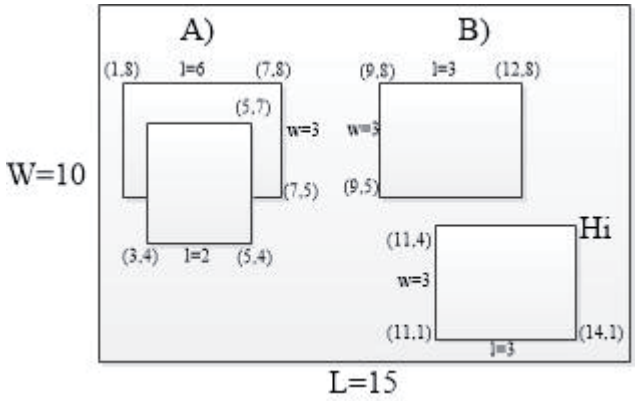

Figura 4.2 Restricción eje Yi.

Caso similar al de la figura 4.2 donde se muestra cómo actúa la restricción del traslapamiento en el eje W en 2 dimensiones. En la figura A se ve claramente como las cajas están sobrepuestas. Si evaluamos la restricción esta no se cumple:

A) $x_i = 1, x_j = 5, y_i = 5, y_j = 7, l_i = 2, w_i = 3, l_j = 3, w_j = 3$

1+6 ≤ 7 v 5+3 ≤ 7

7≤ 5 v 8 ≤ 7 ⇔ F

Por otro lado, las cajas en la figura B cumplen la restricción:

B) $x_i = 9, x_j = 11, y_i = 1, y_j = 5, l_i = 2, w_i = 3, l_j = 3, w_j = 3$

9+3 ≤ 11 v 1+3 ≤ 5

12 ≤ 11 v 4 ≤ 5 ⇔ V

4.4.3 Inclusión

En esta restricción se valida que las cajas estén dentro de las dimensiones del contenedor. Si $p_i$es verdadero, entonces:

- $x_i \geq 0,\ x_i + l_i \leq L$
- $y_i \geq 0,\ y_i + w_i \leq W$
- $z_i \geq 0,\ z_i + h_i \leq H$

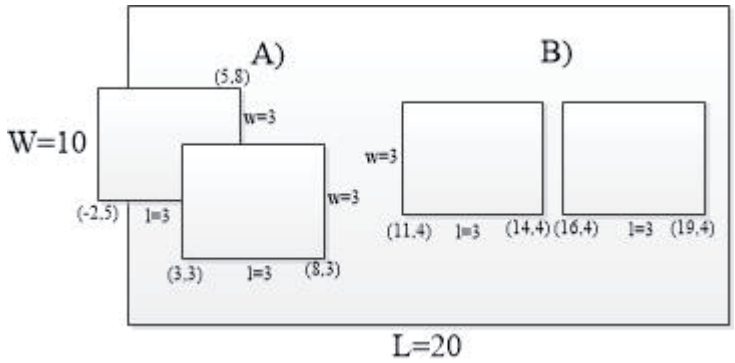

Figura 4.3 Restricción inclusión.

En la figura 4.3 se muestra cómo actúa la restricción de la inclusión de las cajas en el contenedor en 2 dimensiones. Como se puede ver en la figura A se ve claramente como las cajas están fuera del contenedor. Si evaluamos la restricción de inclusión esta no se cumple:

A) $x_i = -2$

$-2 \geq 0 \Leftrightarrow$ F

Por otro lado, las cajas en la figura B cumplen la restricción:

B) $x_i = 11$

$11 \geq 0 \Leftrightarrow$ V

$11+3 \leq 20 \Leftrightarrow$ V

$x_j = 16$

$16 \geq 0 \Leftrightarrow$ V

$16+3 \leq 20 \Leftrightarrow$ V

5 Representación del Problema

Algoritmos de resolución trabajan explorando el espacio de búsqueda. Generalmente construyen una solución desde un estado inicial (contenedor vacío) a un estado final (contenedor con cajas ubicadas en su interior). Las soluciones se generan colocando las cajas o bloques uno a uno dentro del contenedor.

El proceso se puede dividir en 6 aspectos claves identificados en [1]:

- (K1) Representación del espacio residual.
- (K2) Generación de bloques.
- (K3) Selección de espacio residual
- (K4) Selección de bloque
- (K5) Ubicación del bloque en el espacio residual
- (K6) Estrategia general

Cabe destacar que no todos los algoritmos de resolución se componen de estos 6 procesos. Por ejemplo, los trabajos previos a [9] no realizaban el proceso de generación de bloques, por lo que las cajas eran directamente colocadas dentro del contenedor.

A continuación, se detalla las características del algoritmo BS-CLP [3], uno de los algoritmos estado-del-arte para el problema y en el que se basará el proyecto.

5.1 Representación del espacio residual (K1)

El primer elemento (K1) trata la representación del espacio libre, o residual, en el contenedor. Como su nombre lo dice, lo que se intenta representar es el espacio no cargado del contenedor y en general, estos espacios son representados por cuboides los cuales están representados por $R = \{r_1, r_2, r_3, \dots, r_n\}$ que, a diferencia de las cajas, si pueden estar solapados unos con otros.

Para entender mejor esta representación, se propondrá la situación de un contenedor vacío, donde el espacio residual estará definido por r, que, en este caso, será un único cuboide de tamaño idéntico al del contenedor. Como muestra la figura K1, al cargar una caja dentro del contenedor, ya no se tendrá un único espacio residual, sino que se generarán 3 nuevos espacios residuales r_1, r_2 y r_3 (figuras a, b y c respectivamente). Por lo tanto, R representa el espacio libre del contenedor y se actualiza cada vez que una caja es colocada en el contenedor.

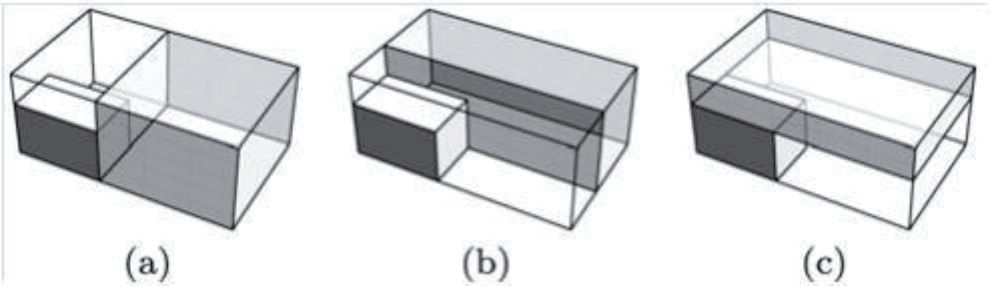

Figura 5.1 Representación de los espacios residuales.

A continuación, se detallan una serie de características y especificaciones para entender en detalle esta representación.

5.1.1 Conjunto R

Formalmente, R se define como un conjunto ordenado de espacios residuales. Se le denomina conjunto ordenado ya que, para beneficio de la resolución del problema, siempre estará ordenado ascendentemente, según la menor distancia de Manhattan, la cual calcula la distancia entre los vértices del espacio y del contenedor (ver más detalles en Sección 5.3).

Cada espacio residual r_i mantiene su ubicación exacta dentro del contenedor. Es decir, por un lado, guarda las coordenadas entre las cuales se encuentra ubicado: xmini, ymini, zmini (coordenada del vértice más cercano al origen) y xmaxi, ymaxi, zmaxi (coordenadas del punto más lejano). Estas coordenadas permiten, por ejemplo, calcular la distancia de Manhattan asociada al espacio. Por otro lado, el espacio residual también guarda sus dimensiones (l_i, w_i, h_i), las que permiten saber si en el espacio r_i caben los bloques que se desean cargar. Note que las dimensiones pueden ser calculadas usando las coordenadas previamente mencionadas (e.g., l_i = xmaxi-xmini).

5.2 Procedimiento de generación de bloques (K2)

Este procedimiento construye bloques mediante la combinación de las cajas disponibles en C, además el procedimiento consta de 2 parámetros que son max_bl que representa la cantidad máxima de bloques y min_fr que es el porcentaje del volumen mínimo ocupado por el nuevo bloque, que tienen los valores de 10.000 y 98% respectivamente. En primer lugar, cada caja genera hasta 6 bloques. Estos bloques se obtienen por todas las rotaciones viables de las cajas (dependiendo de la restricción de orientación). Entonces, un procedimiento iterativo intenta combinar cada bloque con otro colocándolos en contacto a lo largo de los ejes, (ver bucle for en línea 6-8 del algoritmo 1). Dos bloques son combinados sólo si utilizan al menos un 98% del volumen del nuevo bloque.

Los bloques que tengan las mismas dimensiones y que contienen las mismas cajas se consideran sólo una vez. El algoritmo descarta los bloques que no pueden construirse usando las cajas y los bloques cuyas dimensiones exceden el tamaño del contenedor. El proceso se detiene cuando se generan el máximo de bloques o no se pueden generar más bloques diferentes.

5.2.1 Algoritmo para la generación de bloques

A continuación, se muestra el algoritmo que se utiliza para la generación de los bloques.

1: Iniciar el conjunto de los bloques B correspondiente a todas las orientaciones posibles de las cajas de individuales;
2: **while** |B| < max_bl **do**
3: Nuevo bloque N ←∅;
4: **for all** bloque b1 in P **do**
5: **for all** bloque b2 in B **do**
6: **for all** axis in {X, Y, Z} **do**
7: Combina b1 y b2 a lo largo de los ejes para obtener b3;
8: Inserta b3 en N si no es un duplicado y volumen ocupado supera el min_fr**%**
9: **if** N =∅ **then**
10: **break**;
11: **end if**
12: B← B U N y mantiene los primeros max_bl bloques
13: P ←N;
14: **end while**
15: **return** B;

5.2.1.1 Ejemplo de unión de bloques

A continuación, se muestran las formas en las que se pueden unir los bloques en el algoritmo antes visto (línea 7 del algoritmo). Al unir los bloques en forma horizontal (ver figura 5.2), se tendrá como resultado un bloque que tendrá como dimensiones la suma de los largos de ambas cajas y la altura y ancho mayor de los bloques. En los otros ejemplos (figura 5.3 y figura 5.4) se manejan de manera similar las dimensiones, sumando las dimensiones de acuerdo a la forma en la que se desee unir las cajas y manteniendo la mayor longitud con respecto a la otra dimensión en cuestión.

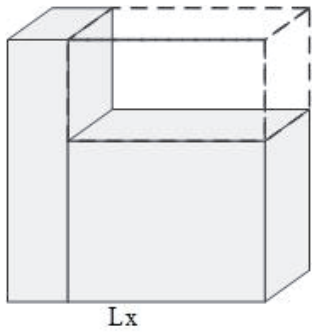

Figura 5.2 Unión horizontal de bloques

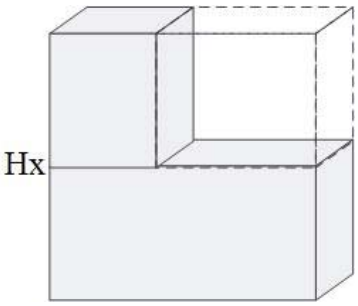

Figura 5.3 Unión vertical de bloques

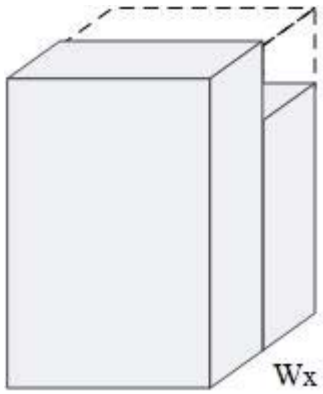

Figura 5.4 Unión ancho de bloques

5.3 Selección de espacio residual por Manhattan (K3)

Ya teniendo la representación de espacios residuales y la de los bloques (generados), ahora se abordará la elección del espacio residual en donde se cargará el siguiente bloque.

5.3.1 Elección del espacio residual a cargar

El espacio prioritario para cargar un bloque será el primer elemento del conjunto R. Cabe mencionar, que los espacios dentro de R se encuentran ordenados ascendentemente de acuerdo a la *menor distancia de Manhattan* entre un vértice del espacio y el vértice más cercano del contenedor.

5.3.1.1 Distancia de Manhattan

La distancia de Manhattan se puede definir como la distancia que se recorrería para llegar de un punto a otro si se siguiera una trayectoria de cuadrícula. En otras palabras, la distancia de Manhattan entre un punto (x, y) y el origen $(0,0)$ de un plano bidimensional, estaría dada por $[(x-0)+(y-0)]$ (ver figura 5.5). Para efectos del problema de carga de contenedores, cuyo plano es tridimensional, la distancia funciona de una manera similar.

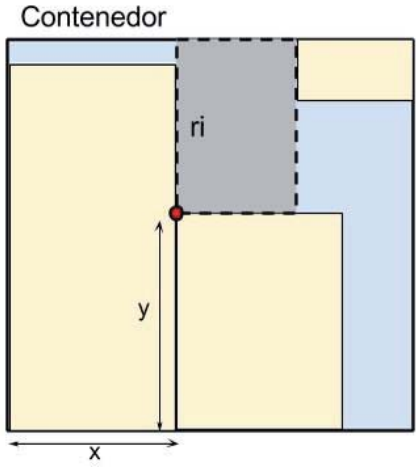

Figura 5.5 Distancia Manhattan

En la figura 5.5 se muestra gráficamente el cálculo de la distancia de Manhattan entre el vértice inferior-izquierdo del espacio residual $\boldsymbol{r_i}$ y el vértice inferior-izquierdo del contenedor.

Un cuboide o espacio residual tiene ocho esquinas, cada una de ellas es asociada a su esquina correspondiente del contenedor. Es decir, la esquina más baja, más a la derecha y más profunda del cuboide se asocia con la esquina símil del contenedor. Matemáticamente la distancia Manhattan entre la esquina del cuboide de coordenadas (x_1, y_1, z_1) y su correspondiente esquina del contenedor con coordenadas (x_2, y_2, z_2), se calcula $|x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2|$.

5.4 Selección del bloque (K4)

La función de evaluación f(b,r)=V(b)-Vloss(b,r) (V-menos.vloss) se utiliza para calificar los bloques que se cargan en un espacio residual r (primer espacio del conjunto R). V (b) corresponde al volumen utilizado del bloque b y Vloss(b,r) es una estimación del volumen desperdiciado en el espacio residual r. La estimación se basa en el hecho de que el máximo espacio utilizable en cada eje de un paralelepípedo debe ser una combinación lineal de las dimensiones de las cajas restantes. Si el bloque b no cabe en el cuboide r, la función devuelve -∞.

Tomando en cuenta que las dimensiones de b son l(b), w(b) y h(b) y las dimensiones de r son l(r), w(r) y h(r). Considerando también que Lmáx es la combinación lineal máxima de la caja restante y que satisface la restricción de capacidad, es decir, Lmax<l(r)-l(b). Análogamente, Wmax y Hmax son las combinaciones lineales máximas relacionadas con la anchura y la altura respectivamente. Finalmente, Vloss =V(r)-(l(b)+Lmax) * (w(b)+Wmax) * (h(b)+Hmax), donde V(r) es el volumen de las cajas contenidas dentro del bloque (ver ejemplo en figura 5.6)

Para calcular los máximos (Lmax, Wmax y Hmax), el problema es modelado como un problema de la mochila. Se resuelve el problema usando programación dinámica con complejidad temporal pseudo-polinomial. El algoritmo de resolución retorna un conjunto de soluciones óptimas para todos los valores posibles relacionados con la restricción de capacidad. El algoritmo de resolución se ejecuta sólo una vez para cada estado: la primera vez que f(b,r) necesita ser evaluado.

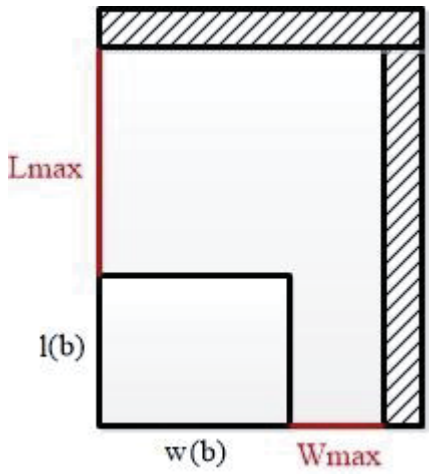

Figura 5.6 Ejemplo Vloss

La figura 5.6 muestra un ejemplo gráfico en 2D de la función Vloss. Al agregar la caja en el espacio residual r, Vloss calcula Lmax y Wmax que representan el máximo espacio que puede ser ocupado por la combinación de cajas restantes. Por lo tanto, el espacio achurado corresponde a una estimación del espacio que se *perderá definitivamente* al agregar la caja b. Este espacio achurado se calcula restando al área de r, el área del rectángulo de dimensiones l(b)+Lmax y w(b)+Wmax. El cálculo es análogo para el problema en 3 dimensiones.

5.4.1 Pseudocódigo algoritmo V-menos-Vloss

A continuación, se muestra el pseudocódigo de la función V-menos-Vloss que retorna el volumen del bloque menos el espacio estimado que se pierde al cargar un bloque:

V-menos-VLoss (Bloque b, Espacio s): retorna un entero

/* Calcula el largo total entre el largo del bloque y Lmax */

RealL ← l(b) + Lmax[l(s) – l(b)]
RealW ← w(b) + Wmax[w(s) – w(b)]
RealH ← h(b) + Hmax[h(s) – h(b)]

/* Finalmente se retorna el volumen ocupado por el bloque menos el volumen estimado que se perdería si se coloca (volumen en rojo) */

return b.volu_ocupado - (Lmax[L]*Wmax[W]*Hmax[H] - RealL*RealW*RealH)

Lmax[i] → Arreglo en donde cada elemento Lmax[i] representa la máxima suma posible de largos de las cajas restantes menor a i. El arreglo se obtiene en tiempo pseudo-polinomial resolviendo el problema de la mochila.

5.5 Proceso de carga del bloque en el contenedor (K5)

Una vez seleccionado, el bloque de acuerdo a la función de evaluación, y el espacio residual r_i donde se cargará el bloque (*menor* distancia de Manhattan), el bloque es colocado en el vértice del espacio correspondiente a esta distancia (*vértice ancla*).

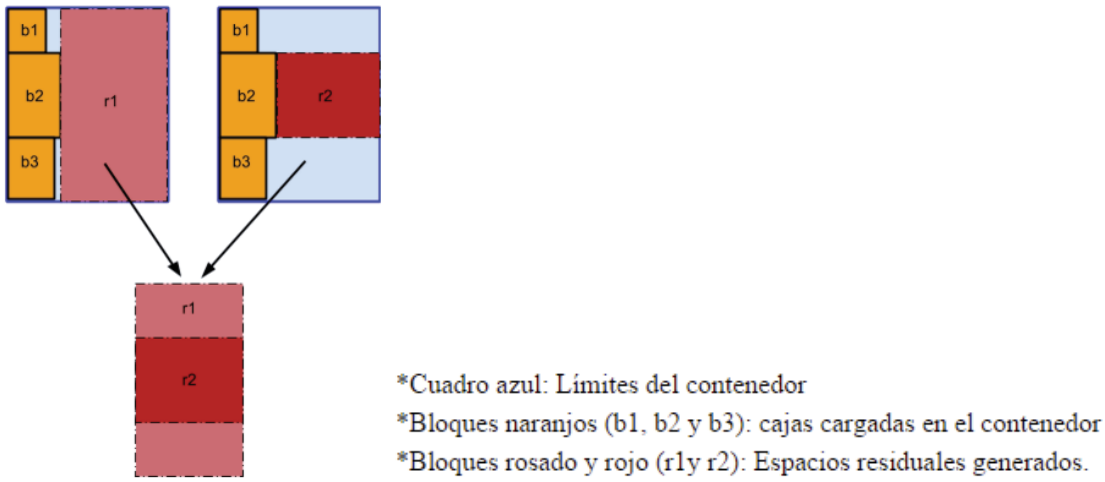

Figura 5.7 Espacio residual maximales.

El proceso que se realiza luego de la selección del bloque es explicado a continuación:

1. El bloque se coloca en el vértice ancla del espacio residual, lo que genera hasta 3 subespacios residuales (como se explicó anteriormente) que son agregados en una lista auxiliar L. El subespacio r_i es eliminado de R.
2. Cada espacio residual que intersecta el nuevo bloque es eliminado de R y genera nuevos subespacios (hasta 6) los que también son agregados en la lista L.
3. Los espacios en la lista L son procesados por un algoritmo que genera una nueva lista L' con *espacios residuales maximales*.
4. Los espacios maximales de la lista L' son agregados al conjunto ordenado R (actualización de R).

Espacio maximal: Es un espacio residual tal, que no puede existir otro espacio residual más grande que lo contenga (ver figura 5.7). Siendo b_1, b_2 y b_3 bloques cargados en el contenedor y r_1 y r_2 espacios residuales generados, se puede apreciar que r_2 está totalmente contenida en r_1. En este caso, dentro del conjunto R se manejaría solo r_1, ya que r_2 no es maximal.

5.6 El procedimiento Beam Search (K6)

El Beam Search (BS) es un algoritmo derivado del Branch-and-Bound que realiza una búsqueda en anchura, e incorpora una heurística para escoger en cada nivel solo los mejores W nodos. Los mejores nodos se guardan en una lista W suele denominarse beam width. Este método sacrifica completitud a cambio de un enfoque heurístico más eficiente.

El algoritmo 2 describe el BeamSearch usado para resolver el problema CLP. BeamSearch recibe como entrada el contenedor Γ, el conjunto de cajas de carga C, el conjunto de bloques B y dos parámetros definidos por el usuario.

En primer lugar, se construye S_0 que es el estado inicial. En el estado inicial (nodo raíz del árbol), la lista de espacios residuales R sólo tiene un elemento: el contenedor. S mantiene la lista de estados o nodos en el nivel actual del árbol.

En cada iteración, los nodos del nivel actual se expanden (es decir, se generan hijos). Cada nodo del nivel actual genera un subconjunto de hijos succ (w^2 hijos para el nodo raíz, y w para el resto de los nodos) y se ponen en S'. Los w estados más prometedores, según una evaluación heurística, dada por un algoritmo voraz (greedy), se conservan para el siguiente nivel (si |S'|<=w, se conservan todos los nodos). Básicamente, para cada nodo/estado s, el algoritmo voraz genera una solución completa y retorna el volumen total ocupado por las cajas en esta solución. Sbest mantiene la mejor solución completa encontrada hasta el momento por el algoritmo voraz.

BeamSearch (input: Γ, C, B, w, input-output: Sbest).

```
R ← { Γ }
S0← estado_inicial; (R,C,B)
while S ∅ do
 Lista de los sucesores S '← ∅
        for all s S do
                if s=S0 then
                        S′←S′ (expand(s, w^2))
                else
                S′←S′ (expand(s,w))
                end if
        end for
 S←se seleccionan los w nodos de S' que maximizan la evaluación del Greedy
 end while
```

El algoritmo beam search termina cuando no se pueden generar más sucesores, en otras palabras, cuando ningún espacio residual tiene las dimensiones para albergar otra caja

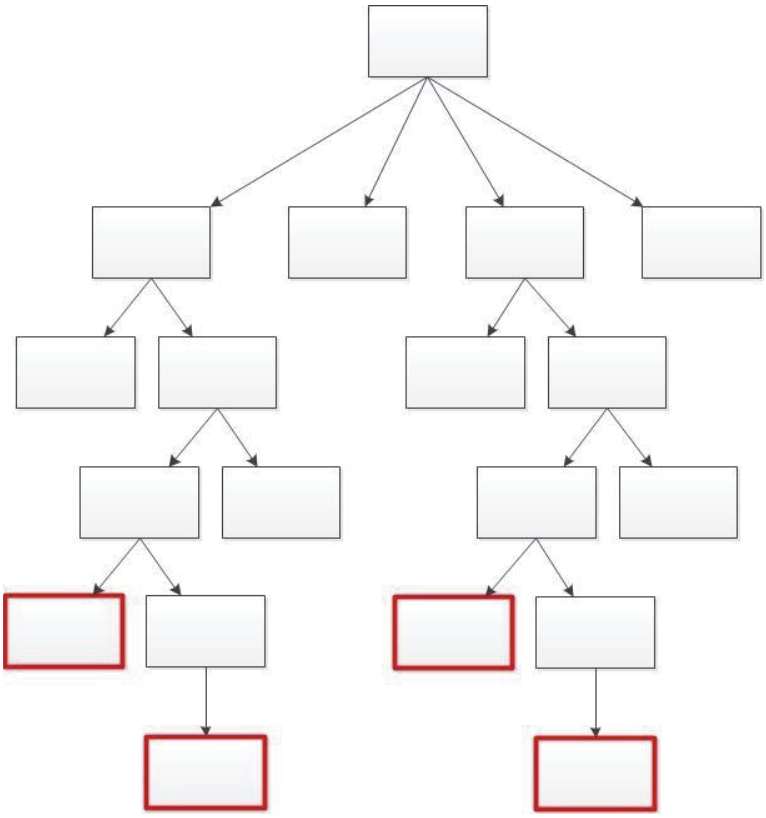

Figura 5.8 Ejemplo beam search

La figura 5.8 muestra un ejemplo del algoritmo beam search con w=2 y sus 4 mejores soluciones marcadas en rojo.

5.6.1 Algoritmo Voraz (función de evaluación)

A continuación, se explica el funcionamiento del algoritmo voraz usado para evaluar los nodos dentro del beam search:

Considerando las siguientes variables:
C: conjunto de cajas disponibles
B: conjunto de bloques que se pueden formar con cajas en C
R: conjunto de espacios residuales

El algoritmo voraz realiza los siguientes pasos:

1. Comienza con una solución o estado parcial (contenedor a medio llenar).
2. Se selecciona el espacio residual r que minimiza distancia Manhattan.
3. Se selecciona bloque b de B más conveniente que quepa en r V-menos-Vloss.
    a. Si no hay bloque que quepa en r, se elimina r de R y vuelve a 2.

    Se coloca el bloque b en r (anchor corner). Se actualiza C, B y R.
    Vuelve a 2 mientras existan espacios residuales.
4. El algoritmo retorna el volumen ocupado por la solución generada.

5.6.2 Estrategia general (doble esfuerzo de búsqueda)

BSG-CLP recibe como entrada el contenedor Γ, el conjunto de cajas de carga C y dos parámetros definidos por el usuario. Estos parámetros son utilizados por el método GeneralBlockGeneration que crea la lista de bloques que se utilizará para la construcción de soluciones. Una vez que los bloques se generan, se aplica el algoritmo beam search varias veces hasta que se alcanza el tiempo límite (timeout). sbest mantiene la mejor solución encontrada hasta el momento en la búsqueda. Además de los elementos de Γ, C y B, relacionados con el estado inicial del problema, el procedimiento BeamSearch utiliza también como entrada el parámetro w. Este parámetro está relacionado con el esfuerzo de búsqueda. w limita el número de llamadas al algoritmo voraz que domina el tiempo de ejecución del beam search: como máximo se realizan w^2llamadas al algoritmo voraz en cada nivel del árbol de búsqueda. w es inicializado en 1 y su valor aumenta a $\lceil\sqrt{2w}\rceil$ después de cada ejecución del beam search. De esta forma, una ejecución del beam search realiza alrededor del doble de llamadas al algoritmo voraz que durante su ejecución previa. Este mecanismo se llama doble esfuerzo de búsqueda (double search effort) y ha sido usado en otros acercamientos constructivos [4, 1,20].

BSG-CLP (input: Γ, C, max_fr, max_bl,timeout).
B ← GeneralBlockGeneration(Γ,C, max_fr, max_bl)
w ← 1; Sbest ← null
while time<timeout do
    s ← BeamSearch(Γ, C, B, w,Sbest)
    w ← $\lceil\sqrt{2w}\rceil$ // doble mecanismo de esfuerzo
end while
return Sbest

6 Problemas detectados

Ya realizado un estudio de las etapas y funciones del algoritmo BSG-CLP, se plantearán algunas observaciones que ayudarán a detectar falencias en la resolución de este problema y también ayudarán a implementar soluciones al respecto:

- Actualmente, el algoritmo al escoger un bloque y cargarlo en el contenedor, estima el máximo espacio que puede ser ocupado, mediante la combinación de cajas restantes (como se explicó anteriormente en sección 2.4), lo que no es necesariamente el espacio que se perderá al cargar un bloque en el contenedor.

- Con el supuesto de querer cargar un camión con planchas y cajas. Claramente es más conveniente poner primero las planchas, de modo que el espacio restante (residual) sea lo menos acotado posible, para cargar las cajas que resten más libremente, esta observación se tratará de incluir en el algoritmo para evaluar beneficios.

- Con respecto a la carga del bloque evaluar la conveniencia de priorizar la carga de bloques voluminosos. En vez de priorizar el abarcar la mayor parte del espacio del contenedor, priorizar bloques de dimensiones lo más heterogéneas posible, para que la carga del contenedor se torne más ordenada.

- La carga de un bloque, puede generar un espacio residual "incómodo" de llenar, ya que actualmente el algoritmo carga bloques de gran tamaño en la mayoría de los casos. Es por esto que los espacios que se generan posterior a la carga del bloque, están más expuestos a quedar como pérdida de espacio, ya que al ser espacios más reducidos y "deformes", al probar cajas o bloques que quepan, tendrá menos probabilidades de aprovechar su espacio en comparación con un espacio cúbico más grande. Se propone evaluar la elección del bloque, para no generar los espacios descritos anteriormente.

- Actualmente el algoritmo conforma bloques aleatoriamente, por lo que no resguarda cajas pequeñas para la última etapa del proceso de carga. Estas cajas pequeñas pueden ofrecer una serie de ventajas, ya que al momento de llenar esos espacios pequeños serán las únicas capaces de hacerlo; por otro lado, estas cajas proveerán más combinaciones posibles para llenar los espacios restantes en el contenedor.

- Selección del espacio residual donde se cargará el siguiente bloque. Actualmente esta elección está dada por el vértice ancla, lo que indetermina qué tan beneficioso puede ser para efectos de la búsqueda de la solución.

7 Propuestas

Se observa la conveniencia de colocar primero bloques planos en vez de los bloques cúbicos. Un ejemplo claro sería el supuesto de querer cargar un camión con planchas y cajas. Claramente es más conveniente poner primero las planchas, de modo que el espacio restante (residual) sea lo menos acotado posible, para cargar las cajas que resten más libremente. Se postularon 2 soluciones que mitigan de una u otra manera esta idea, que son Max_area y hkombat, las cuales privilegian la carga del contenedor con bloques preferentemente planos, y serán detalladas más adelante.

7.1 Evitar la generación de espacios "incómodos"

También se observa que, al momento de cargar una caja, el algoritmo BSG-CLP original selecciona el bloque más conveniente (en la mayoría de los casos uno que maximiza el volumen) y lo pone en el vértice ancla del espacio seleccionado (tal como lo explica el algoritmo en los puntos anteriores), pero esto genera una situación que no favorece.

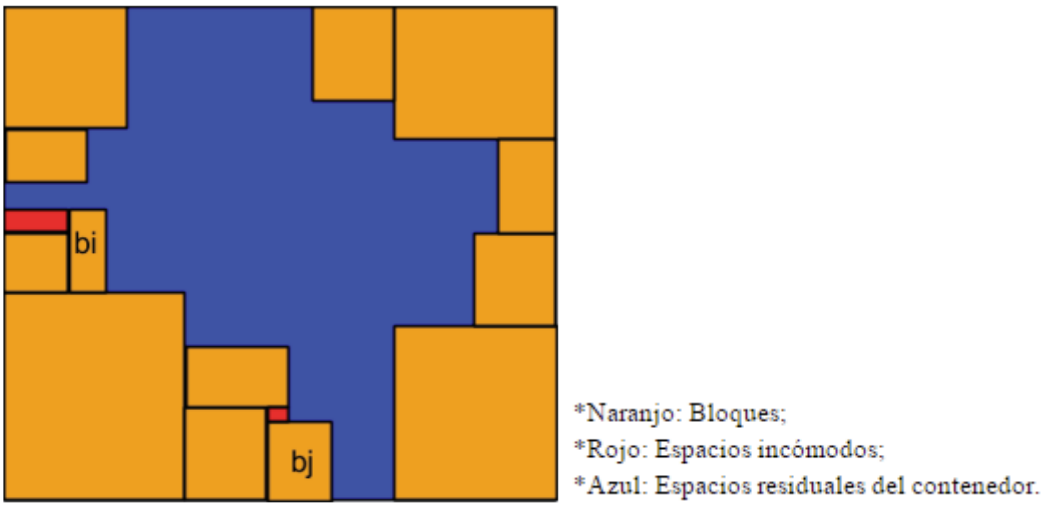

Figura 7.1 Formación de espacios incómodos.

Tal como lo muestra la figura 7.1, la carga del bloque bi en la posición seleccionada, puede generar un espacio residual "incómodo" de llenar (espacio rojo), de igual manera que con la carga del bloque bj. Estos espacios (rojos) están más expuestos a quedar como pérdida de espacio, ya que al ser espacios más reducidos y "deformes", al probar cajas o bloques que quepan, tendrá menos probabilidades de aprovechar su espacio en comparación con un espacio cúbico más grande.

Con el objetivo de evitar la conformación de espacios residuales incómodos de llenar, se propone hkombat. hkombat es un algoritmo que propone una serie de cambios a la función de evaluación, siendo el más importante el cambio de enfoque con respecto a la elección del bloque:

- ❖ trabaja con áreas en vez de volúmenes
- ❖ trata de evitar la conformación de espacios incómodos.

7.2 hkombat

Como anteriormente se mencionó, hkombat es un algoritmo que en vez de colocar los bloques más voluminosos el contenedor, busca realizar una carga prioritaria, evaluando el bloque más conveniente. Este cambio de enfoque viene de la mano, por un lado, con el cambio de paradigma de elección (de volumen a superficie) y con la idea de evitar la conformación de espacios incómodos de llenar (figura 7.2).

El cambio de enfoque que da hkombat, está dado por el cambio en la evaluación, donde ya no se considera el volumen del bloque a cargar, si no que calcula las superficies solapadas de este tanto con el contenedor, como con sus bloques adyacentes, para luego evaluar si es que es conveniente cargarlo.

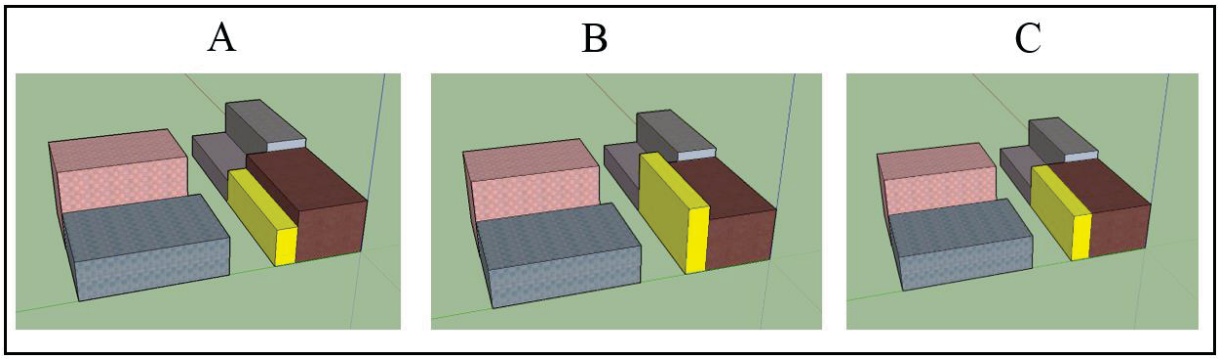

Figura 7.2 Heurística hkombat

Por ejemplo, en la figura 7.2 A y B, el bloque a cargar (amarillo) sería menos conveniente que el de la C. Esto es debido a que la idea de ir ordenando la carga del contenedor, conlleva a evitar la generación de los espacios incómodos de llenar (espacios potencialmente perdidos), y para esto es necesario tratar de acotar el espacio residual donde se cargará el siguiente bloque. Cabe recordar que los espacios residuales se forman a medida que se van colocando bloques y, por ende, son estos mismos bloques los que determinan uno u otro espacio. Es por todo lo anteriormente mencionado que nace la idea de acotar el espacio residual a cargar, para que el bloque idóneo a cargar sea del mismo tamaño de sus bloques adyacentes. Note que en el caso de C es la que posee la mayor cantidad de superficie cubierta con respecto a sus bloques adyacentes y contenedor, es por esto que se privilegiará. Esta preferencia es justamente la que propone la función de evaluación de hkombat la que considera la diferencia entre la suma de superficies cubiertas de los bloques (bloque a colocar, bloques adyacentes y contenedor) menos la superficie cubierta del bloque (2*sup_cubierta-sup_descubierta).

hkombat intenta simular el razonamiento humano, por un lado, comparando bloques a cargar de acuerdo a sus áreas solapadas y por otro lado penaliza los bloques que sobrepasan o quedan por debajo del límite o los márgenes de sus bloques adyacentes para evitar crear espacios incómodos.

A continuación, se detalla el pseudocódigo de este algoritmo, con su cambio de enfoque y el detalle de sus procedimientos:

La función eval_hkombat calcula las superficies cubiertas y descubiertas del bloque a cargar, con respecto a los bloques adyacentes y al contendor

```
eval_hkombat(Bloque b, Contenedor C) ): retorna valor entero
1  sup_cubierta_total ← 0
2  List bloques_ady ← obtener_bloques_adyacentes(C,b)
3  for bloque b_i de bloques_ady
4      sup_cubierta_total ← sup_cubierta_total + superficie_cubierta(b,b_i)
5      sup_cubierta_total ← sup_cubierta_total +  superficie_cubierta_contenedor(b,C)
6      sup_descubierta ← 2*(l(b)*w(b) + w(b)*h(b) + h(b)*l(b)) - sup_cubierta_total
7  return 2*sup_cubierta_total - sup_descubierta
```

Eval_hkombat recibe como parámetros las variables del bloque (con sus puntos de origen y dimensiones) y el contenedor (Con sus dimensiones y cajas cargadas). */

Para entender el funcionamiento de eval_hkombat es necesario conocer los bloques adyacentes. Obt_Bloq_Ady, es una función que retorna una lista con todos los bloques adyacentes a b (línea 2). Luego para cada bloque adyacente se calcula la superficie cubierta entre este y el bloque a cargar (linea 4). Toda esta actualización del valor se guardará en sup_cubierta_total (línea 5). Luego calcula la diferencia entre su superficie total y la cubierta y así se obtiene la superficie descubierta (línea 6). Finalmente retorna 2*sup_cubierta_total - sup_descubierta que es la ecuación para la evaluación de los bloques a cargar (línea 7).

Superficie_cubierta calcula la superficie solapada entre el cubo a cargar y un bloque adyacente determinado .

```
superficie_cubierta (Bloque b, Bloque b_i)): retorna valor entero

maxX ← min(x(b) + l(b), x(b_i) + l(b_i))
minX ← max(x(b) , x(b_i))
maxY ← min(y(b) + w(b), y(b_i) + w(b_i))
minY ← max(y(b) , y(b_i))
maxZ ← min(z(b) + h(b), z(b_i) + h(b_i))
minZ ← max(z(b) , z(b_i))

if (x(b)+l(b) = x(b_i)) || (x(b_i)+l(b_i) = x(b)):
return (maxY-minY) * (maxZ-minZ)

if (y(b)+w(b) = y(b_i)) || (y(b_i)+w(b_i) = y(b)):
return (maxX-minX) * (maxZ-minZ)

if (z(b)+h(b) = z(b_i)) || (z(b_i)+h(b_i) = z(b)):
return (maxX-minX) * (maxY-minY)
```

Dependiendo de la orientación de los bloques calculará la superficie solapada y la retornará.

Superficie_cubierta_contenedor es una función que calcula superficie solapada entre el contenedor y el bloque a cargar (En el caso de haber).

```
superficie_cubierta_contenedor(Bloque B, Contenedor C): retorna valor entero
    sup=0

    if (x(b) = 0) || (x(b) + l(b) = L(C)):
      sup ← sup + w(b)*h(b)

    if (y(b) = 0) || (y(b) +w(b) = W(C)):
      sup ← sup + l(b)*h(b)

    if (z(b) = 0) || (z(b) +h(b) = H(C)):
      sup ← sup + w(b)*l(b)

    return sup
```

Para esto ve si alguna coordenada del bloque corresponde a 0, L,W y H (límites del contenedor), luego calcula la superficie cubierta de esa cara, y finalmente retorna el cálculo guardado en sup.

7.3 Vloss_kombat

Posterior a la creación y evaluación de hkombat, se observa una mejora de resultados en cuanto a los enfoques anteriores, pero no sustancial. Es por esto que resulta contraproducente el hecho de evaluar hkombat sin la función Vloss (pérdida estimada de espacio al cargar un bloque). Por esta razón se crea Vloss_kombat, que básicamente es hkombat con la inclusión de Vloss. Luego de evaluar con pruebas esta propuesta se concluye que resulta conveniente la inclusión ya que mejora aún más los resultados arrojados por hkombat por sí sola.

Con respecto al pseudocódigo de Vloss_kombat y sus cambios estaría representado de la siguiente manera:

```
    eval_vloss_kombat(Bloque b, Contenedor C) ): retorna valor entero
    sup_cubierta_total ← 0
    List bloques_ady ← obtener_bloques_adyacentes(C,b)
    for bloque b_i de bloques_ady
      sup_cubierta_total ← sup_cubierta_total + superficie_cubierta(b,b_i)
      sup_cubierta_total ← sup_cubierta_total + superficie_cubierta_contenedor(b,C)

      sup_descubierta ← 2*(l(b)*w(b) + w(b)*h(b) + h(b)*l(b)) - sup_cubierta_total
      A ← 2*sup_cubierta_total - sup_descubierta
```

Cálculo de variante de V-Vloss.

```
        RealL ← l(b) + Lmax[ l(s) – l(b)]
        RealW ← w(b) + Wmax[w(s) – w(b)]
        RealH ← h(b) + Hmax[h(s) – h(b)]

        vloss= Lmax[L]* Wmax[W]* Hmax[H] - Real *RealW *RealH;
        B ← (l(b). volu_ocupado/ (l(b).volu_ocupado+ vloss))

    return A * B^beta
```

Note que el pseudocódigo es similar al anterior hasta la asignación A ← 2*sup_cubierta_total - sup_descubierta, donde encapsula hkombat en A. Posteriormente por medio del problema de la mochila explicado anteriormente en el punto 5.4, se calcula la mejor combinación de cajas posibles que se puedan cargar en el contenedor, para calcular finalmente el espacio que se perdería al cargar el bloque (Vloss). Luego realiza la siguiente operación: B ←V/V+Vloss, expresión que intenta medir la fracción de volumen que se perdería al cargar el bloque. Se pueden considerar las siguientes consideraciones:

- V y Vloss son positivos, debido a que los 2 son volúmenes tanto del bloque como del espacio que se pierde.
- B es un valor que varía entre 0.5 y 1.
- En el mejor de los casos el espacio que se perderá (Vloss) será 0. En este caso la expresión queda V/V=1.
- En el peor caso Vloss=V y B=0.5.

Finalmente, Vloss_kombat retorna A*B^β, expresión que engloba las dos funciones (A y B) y calcula una evaluación conjunta del bloque.
\ β es un parámetro que pondera la función B.

7.4 Vloss_largeboxes_first

La idea de Vloss_largeboxes_first, consiste en seleccionar los bloques **priorizando aquellos que están conformados con las cajas más grandes.** De esta forma las cajas más pequeñas serán colocadas en las últimas etapas del beam-search. Estas cajas pequeñas nos dan una serie de ventajas, ya que son capaces de encajar en una mayor variedad de espacios grandes o chicos.

Bajo este razonamiento es que se ve la necesidad de integrar esta idea en la evaluación. Es por esto que se crea Vloss_largeboxes_first, modificación en la función de evaluación que integra este raciocinio en la evaluación del bloque que se cargará. La fórmula generalizada que se usará en experimentaciones estará dada por:

C=1/nb_boxes(b), donde se evaluarán los potenciales bloques a cargar calculando el número de cajas que lo componen. En otras palabras, mientras un bloque se componga de la

mayor cantidad de cajas (nb_boxes(b)), será menor ponderado ya que se compone de muchas cajas pequeñas. Situación contraria ocurre con un bloque que se componga de menos cajas, donde la ponderación será más alta y por ende se puede deducir que se compone de cajas más grandes.

Vloss_largeboxes_first es una composición de todas las ideas anteriormente detalladas, donde finalmente evalúa los bloques según estos 3 valores, A*B^ β *C^ γ. El bloque seleccionado es aquel que maximiza este valor.
\ γ es un parámetro que pondera la función C.

7.5 Fair_Vloss

Fair_Vloss intenta incluir un detalle importante en la función de evaluación actual, que es el cambio en la función de evaluación, específicamente en V-menos-Vloss, en la cual se implementa una "*función de evaluación más justa".* Esta función acota el espacio residual perdido que consideraba el algoritmo anterior, tomando en cuenta sólo la pérdida real. La figura 7.3 A) muestra el espacio inevitablemente perdido (espacio rojo) que considera actualmente el algoritmo, lo que no es necesariamente el espacio que se perderá al cargar el bloque b en el contenedor. Según esta propuesta el espacio inevitablemente perdido está dado por las dimensiones correspondiente a w(b) y h(b) del bloque b, tal como muestra B).

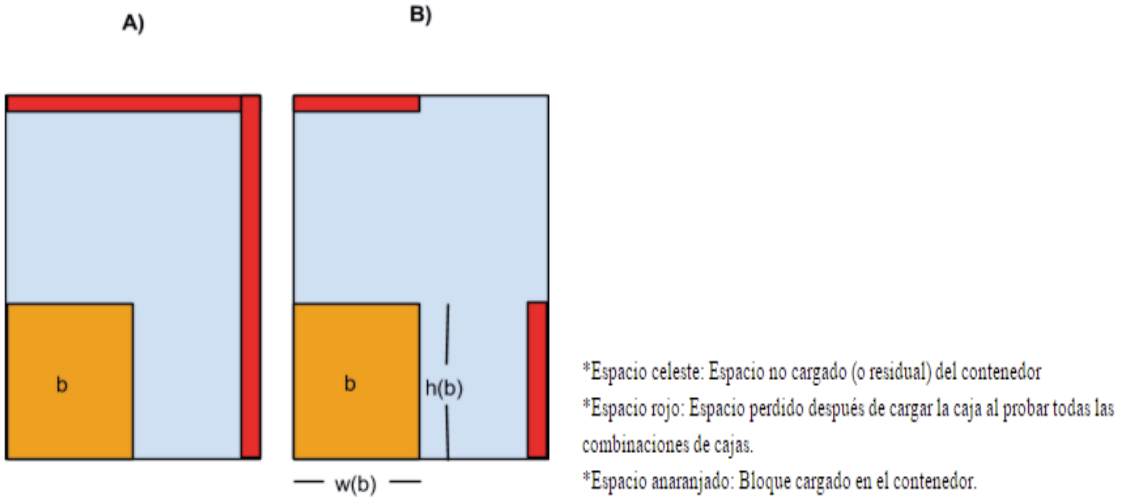

Figura 7.3 Función de evaluación más justa (V-Vloss).

A continuación, se detalla el pseudocódigo de este algoritmo.

```
Fair_VLoss(Bloque* b, Espacio& s)

  L_loss←  max(Lmax[ (L(s))]-( (b.l) + Lmax[ (L(s)-b.l)]),0);
  W_loss←  max(Wmax[ (W(s))] - ( (b.w) + Wmax[ (W(s)-b.w)]),0);
  H_loss←  max(Hmax[ (H(s))] - ( (b.h) + Hmax[ ( H(s)-b.h)]),0);

return b.volu_ocupado - (b.l*b.w* H_loss + b.l* W_loss*b.h + L_loss*b.w*b.h);
```

7.6 Dimensiones ponderadas

Esta heurística trabaja de acuerdo a las dimensiones (largo, ancho y alto) ponderadas de los bloques para elegir el siguiente a cargar. Esto quiere decir que todos los valores de las dimensiones antes de trabajarlos se normalizarán. Básicamente este proceso de normalización consta de ver la proporción de una dimensión, con respecto a la misma del contenedor, pero con 100%=1. Formalmente el cálculo quedaría expresado:

Dimensión bloque / Dimensión contenedor = Dimensión normalizada.

Posterior al proceso de normalización, se realizan una serie de pruebas para verificar en qué grado delta pondera cada una de las dimensiones ponderadas. Finalmente se determina $\delta = 2$.

Un ejemplo de esto se ve en la figura 7.4, donde en el eje Y, el bloque de ancho 5 se compara con el contenedor de ancho 10 y se obtiene la normalización de ésta dimensión que bajo el cálculo resulta 5/10 => 1/2 => 0,5. En el eje X pasa algo similar, donde se comparan los largos tanto del bloque como el del contenedor, resultando la expresión 4/16 => 0,25.

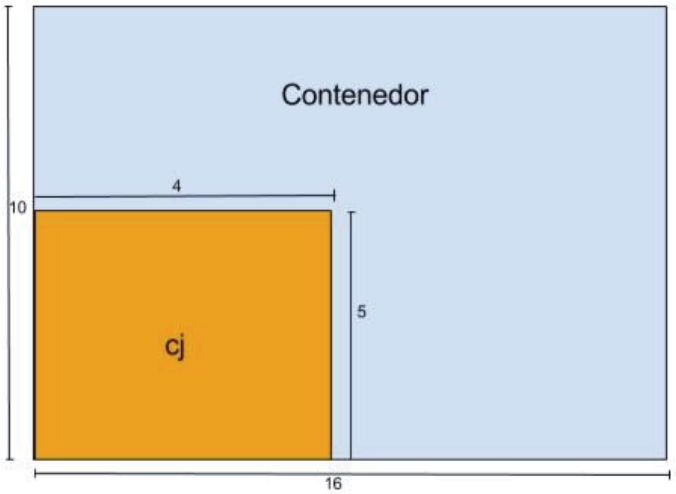

Figura 7.4 Perímetros cuadrados.

Una vez determinadas cada una de las dimensiones normalizadas la función de evaluación retorna rl^δ+rw^δ+rh^δ. Donde rl+rw+rh son el perímetro del bloque y δ es un parámetro que pondera la función.

Cabe destacar que valores altos de delta privilegian la elección de cajas con dimensiones lo más heterogéneas posible como lo son figuras largas, planas o planchas, que son finalmente las que incomodarán o acotarán en menor grado el espacio residual que vaya quedando a lo largo del proceso de carga.

Algorítmicamente, esta función se representa de la siguiente manera:

```
ponderedPerimeter_eval(Bloque* b, Espacio& s, δ)

  if(L(s) – b.l<0 || W(s) – b.w<0 || H(s) – b.h<0)

   return;

  rl←   (b.l) / L(s);
  rw←  (b.w) / W(s);
  rh←  (b.h) / H(s);

  return (rl^δ+rw^δ+rh^δ);
```

8 Experimentos

Los experimentos se realizaron en un servidor PowerEdge T420, con 2 quad-procesadores Intel Xeon, 2.20 GHz y 8 GB de RAM, corriendo Ubuntu Linux. Los códigos se implementaron en C++ y compilado con gcc.

Para evaluar la efectividad de las heurísticas planteadas se hará uso de una serie de 16 conjuntos de problemas (BR0-BR15) utilizados por Franslau y Bortfeldt para analizar un algoritmo propio. Cada conjunto se compone de 100 casos. Los 16 conjuntos de instancias se pueden clasificar en tres categorías: BR0 consta de sólo un tipo de caja (homogénea); BR1-7 consta de unos pocos tipos de cajas (débilmente heterogénea); y BR8-BR15 consta de hasta 100 tipos de cajas (muy heterogénea). Todas las instancias de prueba imponen una serie de restricciones sobre las posibles orientaciones para las cajas individuales. No se imponen restricciones adicionales.

Las distintas funciones de evaluación propuestas se implementaron dentro del algoritmo BSG-CLP. Estas heurísticas se compararon con max_volumen, max_area y V-menos-VLoss. Max_volume y max_area son heurísticas que como su nombre lo indica seleccionan el bloque que maximiza el volumen y el bloque que maximiza su superficie respectivamente.

8.1 Heurísticas que no usan Vloss

En una primera serie de experimentos se compararon max_area, max_volumen, hkombat y ponderedPerimeter_eval que no calculan Vloss. El tiempo para cada prueba fue limitado a 30 segundos. El valor para δ se obtuvo de otra serie de experimentos no reportada en este trabajo.

En la tabla 8.1 se muestran los resultados de los experimentos. Cada fila representa a un conjunto de instancias. Las columnas representan cada una de las heurísticas. Los valores corresponden al volumen ocupado promediado por las soluciones obtenidas.

Tabla 8.1 Experimentos de heurísticas que no usan Vloss con un tiempo de 30s.

	max_area(30s)	max_volumen(30s)	Hkombat(30s)	ponderedPerimeter_eval (30s) δ = 2.0
BR0	**90.33**	90.31	90.29	90.26
BR1	95.18	95.24	**95.30**	95.41
BR2	95.61	95.64	**95.76**	95.82
BR3	95.51	95.80	**95.93**	95.94

	max_area(30s)	max_volumen(30s)	Hkombat(30s)	ponderedPerimeter_eval (30s) ɗ= 2.0
BR4	95.33	95.60	**95.78**	95.87
BR5	95.11	95.38	**95.64**	95.67
BR6	94.93	95.17	**95.58**	95.53
BR7	94.53	94.94	**95.14**	95.12
BR8	93.75	94.26	**94.75**	94.49
BR9	93.52	93.98	**94.53**	94.33
BR10	93.38	93.84	**94.40**	94.14
BR11	93.19	93.65	**94.22**	94.01
BR12	93.06	93.50	**94.12**	93.92
BR13	92.95	93.48	**93.99**	93.76
BR14	92.96	93.31	**93.77**	93.61
BR15	92.90	93.28	**93.77**	93.59
Av. 1-7	95.17	95.39	**95.59**	95.62
Av. 8-15	93.21	93.66	**94.19**	93.98
Av. 1-15	94.13	94.47	**94.85**	94.75

Como se puede apreciar en la tabla 8.1 se ve qué hkombat gana en las instancias desde BR1 hasta BR15, pero en la instancia BR0 pierde contra max_area. Por otra parte, se ve qué hkombat gana en los promedios finales de las instancias.

8.2 Heurísticas que usan Vloss

En otra serie de experimentos se compararon VLoss, Vloss_kombat con β=1.0, Vloss_largeboxes_first con ɣ=0.2 y Fair_vloss, los cuales usan la estimación del volumen perdido (VLoss). Los valores para β y ɣ se obtuvieron de otra serie de experimentos no reportada en este trabajo.

En la tabla 8.2 se muestran los resultados de los experimentos. Cada fila representa a un conjunto de instancias. Las columnas representan cada una de las heurísticas. Los valores corresponden al volumen ocupado promediado por las soluciones obtenidas.

Tabla 8.2 Experimentos de heurísticas que si usan Vloss con un tiempo de 30s.

	V-VLoss(30s)	Vloss_kombat (30s) β=1.0	Vloss_largeboxes_first (30s) ɣ=0.2	Fair_vloss(30s)
BR0	**90.54**	90.47	90.49	90.44
BR1	95.47	95.47	**95.65**	95.43
BR2	96.00	96.09	**96.18**	95.99
BR3	96.24	96.19	**96.31**	96.15
BR4	96.03	96.06	**96.27**	96.02
BR5	95.89	96.01	**96.11**	95.82
BR6	95.81	95.85	**95.99**	95.70
BR7	95.47	95.49	**95.63**	95.34
BR8	94.72	94.96	**95.04**	94.68
BR9	94.47	94.77	**94.81**	94.44
BR10	94.36	94.56	**94.66**	94.23
BR11	94.13	94.36	**94.60**	94.07
BR12	94.08	94.18	**94.33**	94.00
BR13	93.88	**94.13**	94.10	93.84
BR14	93.84	93.89	**93.98**	93.74
BR15	93.78	93.89	**93.89**	93.64
Av. 1-7	95.84	95.88	**96.02**	95.78
Av. 8-15	94.16	94.34	**94.43**	94.08
Av. 1-15	94.94	95.06	**95.17**	94.87

Como se puede apreciar en la tabla 8.2 se ve que Vloss_largeboxes_first gana en las mayorías de las instancias, aunque pierde en algunas se puede ver que gana en los promedios finales de las instancias. También se puede ver que la nueva heurística implementada (Fair_vloos) pierde en todas las instancias, creemos que esto se debe a que al acotar el espacio residual perdido también se acotan las combinaciones lineales por lo que las soluciones encontradas no son las mejores.

8.3 Heurística que usan Vloss (con 150s)

En otra serie de experimentos se compararon BSG-CLP, Vloss_kombat con β=1.0 y Vloss_largeboxes_first con ɣ=0.2. Los dos últimos se corrieron con un tiempo de 150 segundos para poder hacer una comparación con los resultados de BSG-CLP.

En la tabla 8.3 se muestran los resultados de los experimentos. Cada fila representa a un conjunto de instancias. Las columnas representan cada una de las heurísticas. Los valores corresponden al volumen ocupado promediado por las soluciones obtenidas.

Tabla 8.3 Experimentos de heurísticas que si usan Vloss con un tiempo de 150s.

	BSG-CLP (150s)	Vloss_kombat (150s) β=1.0	Vloss_largeboxes_first (150s) ɣ=0.2	BSG-CLP (500s)
BR0	90.91	90.48	90.53	**90.97**
BR1	95.60	95.67	**95.84**	95.69
BR2	96.12	96.31	**96.42**	96.24
BR3	96.32	96.49	**96.60**	96.49
BR4	96.14	96.37	**96.54**	96.31
BR5	96.00	96.31	**96.44**	96.18
BR6	95.88	96.20	**96.34**	96.05
BR7	95.58	95.89	**96.08**	95.77
BR8	95.06	95.45	**95.64**	95.33
BR9	94.82	95.25	**95.39**	95.07
BR10	94.71	95.11	**95.23**	94.97
BR11	94.52	94.91	**95.07**	94.80
BR12	94.36	94.84	**94.92**	94.64

	BSG-CLP (150s)	Vloss_kombat (150s) β=1.0	Vloss_largeboxes_first (150s) ɣ=0.2	BSG-CLP (500s)
BR13	94.34	94.68	**94.80**	94.59
BR14	94.23	94.51	**94.65**	94.49
BR15	94.13	94.41	**94.55**	94.37
Av. 1-7	95.95	96.18	**96.32**	96.11
Av. 8-15	94.52	94.90	**95.03**	94.78
Av. 1-15	95.19	95.49	**95.63**	95.40

Como se puede apreciar en la tabla 8.3 se ve que Vloss_largeboxes_first con 150 segundos mejora sus resultados y le gana a BSG-CLP con 150 segundos y 500 segundos en las mayorías de las instancias, aunque pierde en BR0 se puede ver que gana en los promedios finales de las instancias.

8.4 Comparación de heurísticas

En las tablas 8.4 y 8.5 se muestran comparaciones entre pares de heurísticas para analizar en más detalle los resultados obtenidos. Para todas las tablas se ven los resultados obtenidos de la primera heurística en comparación con la segunda para ir esclareciendo que tan beneficiosa resulta en comparación con la otra. Esta comparación se realiza bajo el método GAP, que ve la diferencia de resultados entre estrategias, bajo un determinado porcentaje.

Por ejemplo en la primera comparación de la tabla 8.4 (20%), se detalla el GAP entre hkombat vs max_area, que refleja cantidad de pruebas en las que hkombat (primera heurística) es un 20%, como mínimo, mejor o peor que max_area, añadiendo una prueba al gana cuando es mejor o pierde en el caso contrario. Así sucesivamente con las otras comparaciones y con tabla 8.5.

Tabla 8.4 Comparación de resultados de experimentos con un 20%.

	hkombat **VS** max_area	hkombat **VS** max_volumen	Vloss_kombat **VS** V-menos-Vloss	Vloss_largeboxes_first **VS** V-menos-Vloss
Gana	507	240	107	164
Pierde	15	34	50	37

Tabla 8.5 Comparación de resultados de experimentos con un 50%.

	hkombat **VS** max_area	hkombat **VS** max_volumen	Vloss_kombat **VS** V-menos-Vloss	Vloss_largeboxes_first **VS** V-menos-Vloss
Gana	16	7	3	5
Pierde	1	3	1	1

Según los resultados arrojados por las tablas 8.4 y 8.5, en las 2 primeras comparaciones (heurísticas que no utilizan Vloss) hkombat obtiene mejores resultados frente a las heurísticas que maximizan el área y volumen, esto se debe a que hkombat realiza una carga prioritaria, evaluando el bloque más conveniente. Este favorable resultado viene de la mano con el cambio de paradigma de elección (de volumen a superficie) y con la idea de evitar la conformación de espacios incómodos de llenar. Por otro lado en las heurísticas que utilizan Vloss también se reflejan buenos resultados tanto en Vloss_kombat como en Vloss_largeboxes_first, siendo esta última la que obtiene exclusivamente mejores resultados, debido a la amplia incorporación de reformulaciones en la heurística.

9 Gráficos de convergencia

A continuación se muestran los gráficos de convergencia, en los cuales se puede visualizar el tiempo que se demora cada instancia en encontrar el óptimo local. Cada instancia se ejecutó con un tiempo de 150 segundos.

En cada gráfico se puede apreciar cuando se van encontrando las soluciones y como estas van mejorando (Puntos azul). Asimismo se puede visualizar el momento en el cuál el árbol de búsqueda generado por Beam Search termina de ser construido y comienza la construcción de uno de mayor tamaño (puntos rojos). Estos puntos también representan los óptimos locales de cada árbol generado.

Por otra parte como se puede ver las instancias que son más homogéneas (Figura 9.1 y Figura 9.2) llegan al punto de convergencia casi de inmediato, en cambio las instancias que son más heterogéneas (Figura 9.3 y Figura 9.4) se demoran más. Esto se debe a que al tener cajas más homogéneas se generan bloques más grandes por lo que se llena más rápido el contenedor y al expandir el árbol de búsqueda no se encuentran mejores. En cambio las cajas más heterogéneas tienen mayores formas de combinar las cajas para llenar el contenedor y al momento de expandir el árbol de búsqueda se exploran estas nuevas formas generando mejores soluciones.

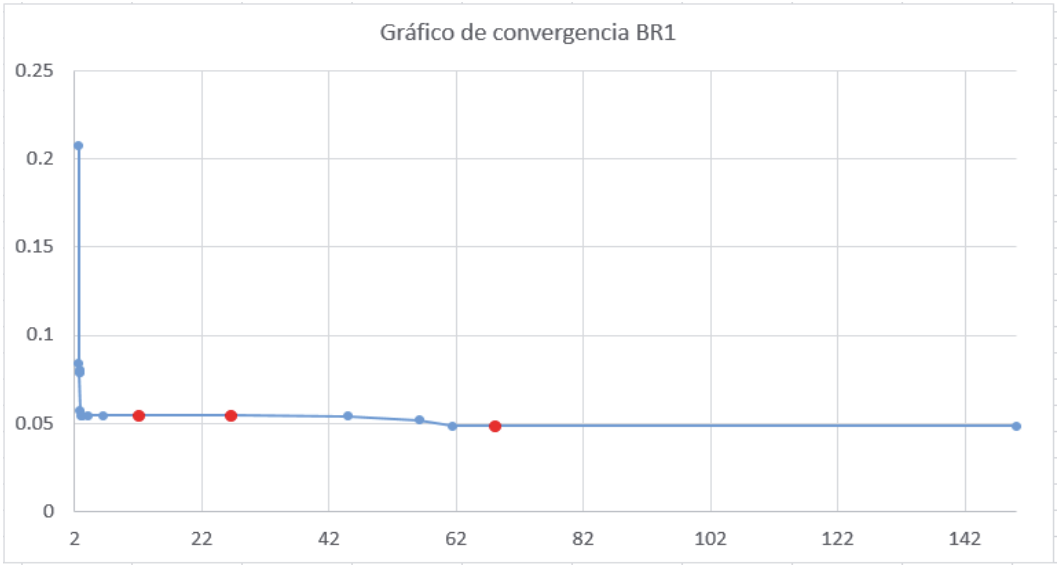

Figura 9.1 Grafico de convergencia BR1.

Figura 9.2 Grafico de convergencia BR6.

Figura 9.3 Grafico de convergencia BR8.

Figura 9.4 Grafico de convergencia BR12.

10 Conclusión

Con todo lo anteriormente mencionado, podemos afirmar que el CLP es un problema muy arraigado en diferentes industrias a nivel mundial, lo que se refleja en la cantidad de trabajos e investigaciones que hay en torno al tema en la actualidad. Encontrar soluciones óptimas que maximicen el volumen ocupado del contenedor es muy costoso en tiempo, y la necesidad de encontrar soluciones rápidas es muy alta, es por eso que las heurísticas cumplen un rol muy importante, ya que encuentran en corto tiempo soluciones "de calidad". A pesar de no generar soluciones óptimas, proveen un importante acercamiento a éstas.

Tras desglosar y estudiar el funcionamiento del BSG-CLP, se plantearon una serie de observaciones que eran obviadas por este algoritmo, tales como el evitar generar espacios incómodos durante el proceso de carga, el resguardo de cajas pequeñas para la parte final de la carga, la propuesta de cambios en la función de evaluación de bloques, etc. Tras implementar los detalles mencionados, dichas estrategias se sometieron a una serie de pruebas con el objetivo de cuantificar el beneficio de su inclusión, arrojando en su mayoría resultados positivos en comparación con otros métodos existentes, siendo Vloss_largeboxes_first la estrategia que mejor resuelve este problema en los tiempos estipulados.

Luego de todas las investigaciones y experimentos realizados, se puede mencionar que uno de los caminos idóneos en la búsqueda de la soluciones a éste problema radica en incorporar o reconsiderar detalles que lógicamente pueden encaminar el proceso de carga, acotando las combinaciones probables de llenado, y así disminuyendo el tiempo en el que se logra llegar a esta solución. Por otra parte, pasando en otro plano, se podría pensar en la incorporación de un nuevo elemento clave para los algoritmos de resolución. Este elemento podría considerar una selección heurística de la estrategia a utilizar de acuerdo al grado de heterogeneidad (u homogeneidad) de las cajas. Para efectos de este proyecto solo se contempló una estrategia que privilegiaba la colocación de las cajas más voluminosas primero, dejando para una etapa final las más pequeñas (Vloss_largeboxes_first).

Posterior a toda la investigación e implementación de distintas formas de carga, se puede mencionar que los algoritmos incompletos de búsqueda se acercan cada vez más al 100% de volumen ocupado del contenedor. Recalcamos que se está tratando con solo algunos componentes clave de este tipo de algoritmos, por lo que los avances que se pueden lograr en torno a este tema aún pueden ser variados si tomamos en cuenta el resto de los componentes como por ejemplo el proceso de creación de bloques o la selección del siguiente espacio residual.

11 Referencias

[1] Wenbin Zhu ·Wee-Chong Oon · Andrew Lim · YujianWeng The six elements to block-building approaches for the single container loading problem. 2012; 37(3):1–15.

[2] Bischoff EE, Ratcliff MSW (1995) Issues in the development of approaches to container loading. OMEGA Int J Manag Sci 23(4):377–390. doi:10.1016/0305-0483(95)00015 G.

[3] I. Araya, M.-C. Riff A beam search approach to the container loading problem. Computers & Operations Research 43 (2014) 100–107.

[4] Fanslau T, Bortfeldt A. A tree search algorithm for solving the container loading problem. INFORMS Journal on Computing 2010;22(2):222–35.

[5] Martello S, Pisinger D, Vigo D. The three-dimensional bin packing problem. Operational Research 2000;48(2):256–67.

[6] Chien CF, Wu WT (1998) A recursive computational procedure for container loading. Comput Ind Eng 35(1–2):319–322. doi:10.1016/S0360-8352(98)00084-9

[7] Bischoff EE (2006) Three-dimensional packing of items with limited load bearing strength. Eur J Oper Res 168(3):952–966. doi: 10.1016/j.ejor.2004.04.037

[8] Dyckhoff H, Finke U (1992) Cutting and packing in production and distribution: a typology and bibliography, 1st edn. Contributions to management science. Physica-Verlag, Heidelberg.

[9] Eley M (2002) Solving container loading problems by block arrangement. Eur J Oper Res 141(2):393–409. doi:10.1016/S0377- 2217(02)00133-9

[10] Bortfeldt A, Gehring H (2001) A hybrid genetic algorithm for the container loading problem. Eur J Oper Res 131(1):143–161. doi:10.1016/S0377-2217(00)00055-2.

[11] Bortfeldt A, Gehring H, Mack D (2003) A parallel tabu search algorithm for solving the container loading problem. Parallel Comput 29(5):641–662. doi:10.1016/S0167-8191(03)00047-4.

[12] Gehring H, Bortfeldt A (1997) A genetic algorithm for solving the container loading problem. Int Trans Oper Res 4(5–6):401–418. doi:10.1111/j.1475-3995. 1997.tb00095. x.

[13] Lim A, Rodrigues B, Yang Y (2005) 3-D container packing heuristics. Appl Intell 22(2):125–134. doi:10.1007/s10489-005-5601-0.

[14] Lins L, Lins S, Morabito R (2002) An n-tet graph approach for non-guillotine packings of n-dimensional boxes into an n-container. Eur J Oper Res 141(2):421–439. doi:10.1016/ S0377-2217(02)00135-2.

[15] Mack D, Bortfeldt A, Gehring H (2004) A parallel hybrid local search algorithm for the container loading problem. Int Trans Oper Res 11(5):511–533. doi:10.1111/j.1475-3995.2004.00474. x.

[16] Parreño F, Alvarez-Valdes R, Oliveira JE, Tamarit JM (2010) Neighborhood structures for the container loading problem: a VNS implementation. J Heuristics 16(1):1–22. doi:10.1007/ s10732-008-9081-3.

[17] Pisinger D (2002) Heuristics for the container loading problem. Eur J Oper Res 141(2):382–392. doi:10.1016/S0377- 2217(02)00132-7.

[18] Ratcliff MSW, Bischoff EE (1998) Allowing for weight considerations in container loading. OR Spektrum 20(1):65–71. doi:10.1007/BF01545534.

[19] Terno J, Scheithauer G, Sommerweiß U, Riehme J (2000) An efficient approach for the multi-pallet loading problem. Eur J Oper Res 123(2):372–381. doi:10.1016/S0377-2217(99)00263-5.

[20] Zhu W, Lim A. A new iterative-doubling greedy-lookahead algorithm for the single container loading problem. European Journal of Operational Research 2012;222(3):408–17.

[21] Che CH, Huang W, Lim A, Zhu W (2011) The multiple container loading cost minimization problem. Eur J Oper Res 214(3):501– 511. doi: 10.1016/j.ejor.2011.04.017,