Universidad Peruana de Ciencias Aplicadas

Escuela de Ingeniería de Sistemas y computación

Trabajo parcial de Complejidad Algorítmica

Docente

* Luis Martín Canaval Sánchez

Integrantes:

* Willington Jesus Ortiz Maurtua [ u20161c963 ]
* Diego Alonso Castillo Ghersi [ u201716002 ]
* Joaquín Alejandro García Guzmán [ u201713735 ]

Lima - 2019

Índice

1. Introducción ……………………………………………………………………………………….. 03
2. Objetivos …………………………………………………………………………………………… 04
3. Marco teórico ……………………………………………………………………………………... 05
4. Análisis de complejidad …………………………………………………………………………. 06
5. Experimentación…………………………………………………………………………………… 13
6. Conclusiones ……………………………………………………………………………………… 18
7. Referencias ……………………………………………………………………………………….. 19
8. Introducción

El problema de empaquetamiento tiene como finalidad empaquetar varios objetos de diferentes tamaños en un contenedor, con el principal objetivo de que minimizar las zonas sin utilizar, es decir, el espacio perdido o desperdiciado.

Este problema afecta a diversos campos de la industria, como el transporte de materia prima a través de vagones en un tren, el almacenamiento de recursos en el almacén de una fábrica, almacenamiento de archivos de computadora con tamaños específicos en bloques de memoria de tamaño fijo, entre otros.

Debido al amplio número de sectores que son afectados por este problema, el presente trabajo tiene como finalidad el análisis e implementación de tres algoritmos de empaquetamiento. De los cuales, se analizará su complejidad, tiempo de ejecución y consumo de memoria. Asimismo, se implementará una interfaz gráfica para la visualización de estos.

1. Objetivos

El objetivo principal del presente trabajo es realizar un análisis e implementación de tres algoritmos de empaquetamiento, probarlos en diferentes escenarios y generar conclusiones y recomendaciones a partir de los resultados.

Los objetivos específicos que se han considerado son los siguientes:

* Implementación de un algoritmo que permita la generación de casos de prueba aleatorios para la evaluación de los algoritmos.
* Análisis de las ventajas y desventajas de los algoritmos implementados, en diversos escenarios, y de su complejidad algorítmica
* Implementación de tres algoritmos de empaquetamiento.
* Implementación de una interfaz gráfica que permita la visualización de los datos de salida.

1. Marco teórico
   1. Heurística

Los algoritmos heurísticos son aquellos que encuentran una solución, sin embargo no pueden reconocer si es una solución óptima o no. Un punto a favor de estos, son sus bajos tiempos de cálculo a comparación de los métodos exactos, lo que permite que se consideren una estrategia de solución óptima para problemas con grandes cantidades de datos. Los algoritmos heurísticos se separan en dos tipos, los constructivos y los de mejora.

* + 1. Heurística Constructiva

La heurística constructiva se basa en crear soluciones a partir de agregar los elementos uno por uno. Por ello, solo se puede obtener la solución al final del algoritmo. Dentro de estos, se pueden dividir los que se basan por el posicionamiento, por el estado físico, por niveles y por el perfil.

* + 1. Heurística de Mejora

La heurística de mejora empieza a partir de una solución inicial, esta puede ser obtenida de forma aleatoria o a través de una heurística constructiva. Desde la solución inicial se aplican pequeños cambios para ir mejorando la calidad de esta, hasta que cumpla una condición de parada. Dentro de estos se encuentra los que lo realizan sobre secuencias y los que lo realizan en el diseño.

* 1. Algoritmos de fuerza bruta

Consisten en enumerar sistemáticamente todos los posibles candidatos para una solución y verificar si cada uno de estos soluciona el problema. La solución de un problema con un algoritmo de fuerza bruta depende únicamente del poder de computación del dispositivo ejecutandolo y generalmente son los tipos de algoritmos que demoran más en encontrar una solución.

* 1. Algoritmos divide y vencerás

Los algoritmos divide y vencerás se caracterizan por dividir un problema en partes más pequeñas del mismo, se realizan tantas divisiones como sea necesario hasta llegar a partes donde la solución sea obvia o muy fácil de calcular. Al final, la solución al problema principal se genera a partir de las soluciones encontradas.

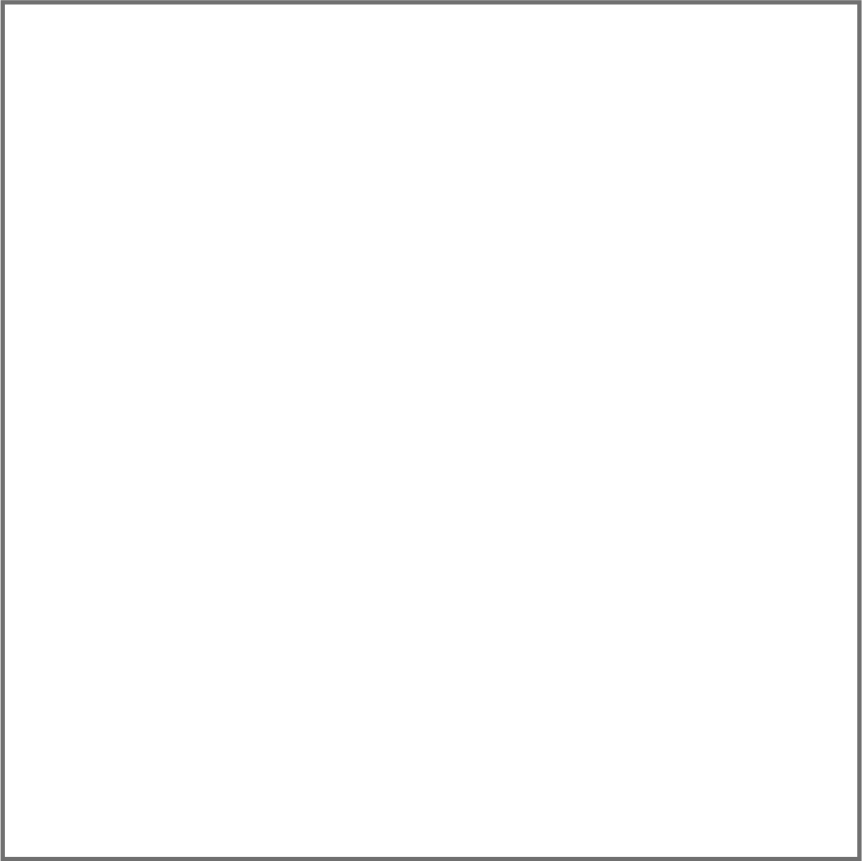
1. Algoritmos
   1. Binary Tree Bin Packing

**Estrategia**

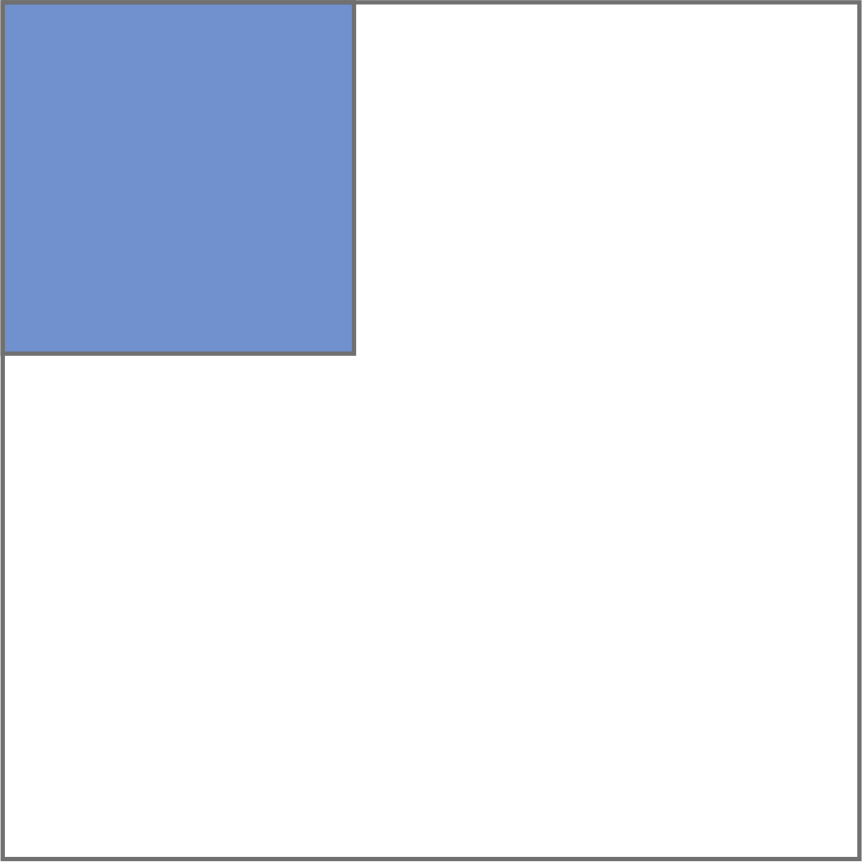
Este es un algoritmo heurístico constructivo que aplica la estrategia de divide y vencerás, ya que divide la plancha en zonas más pequeñas con el objetivo de agregar cajas y minimizar la pérdida del espacio disponible.

**Funcionamiento**

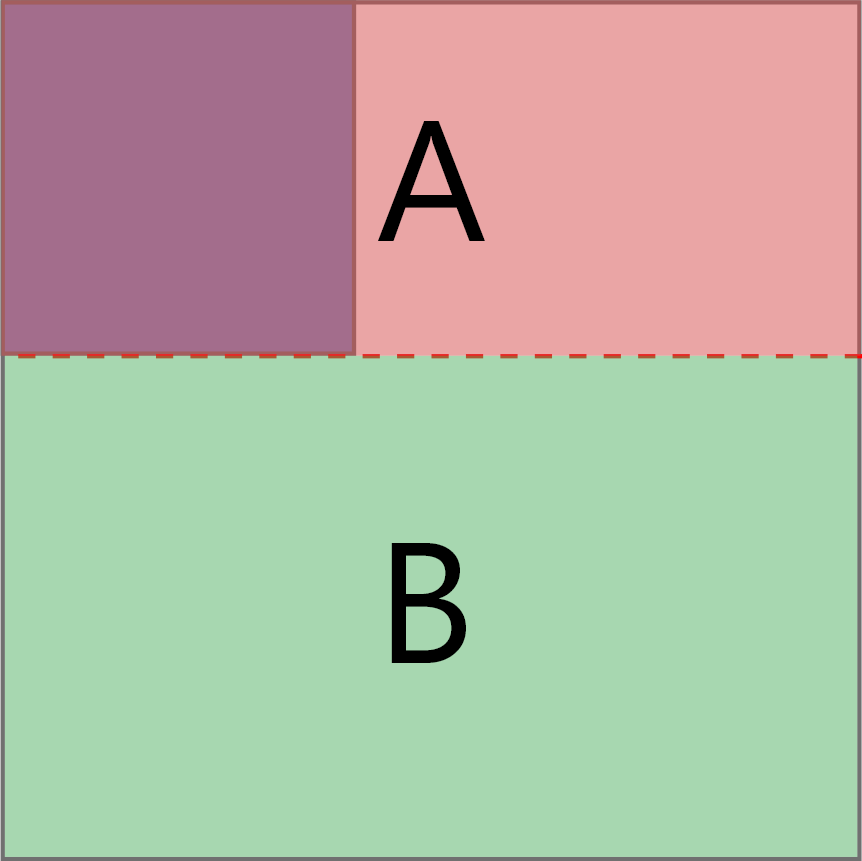
1. El algoritmo inicia con la plancha vacía que será utilizada para el almacenamiento de las cajas.



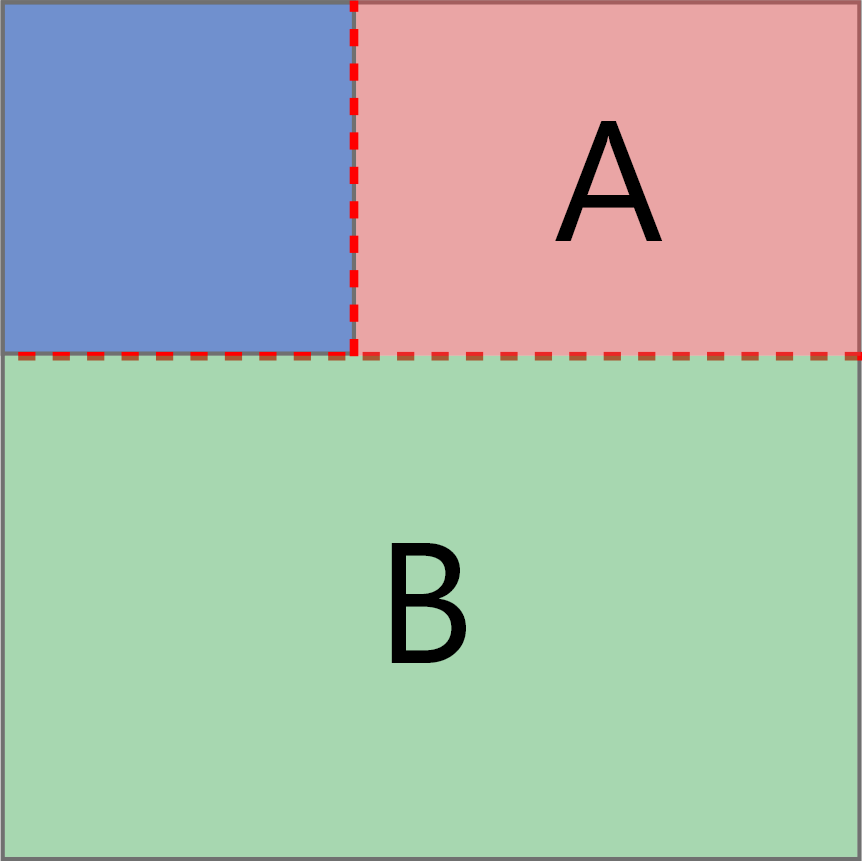
1. Cuando ingresa la primera caja, ésta se colocará en la esquina superior izquierda.



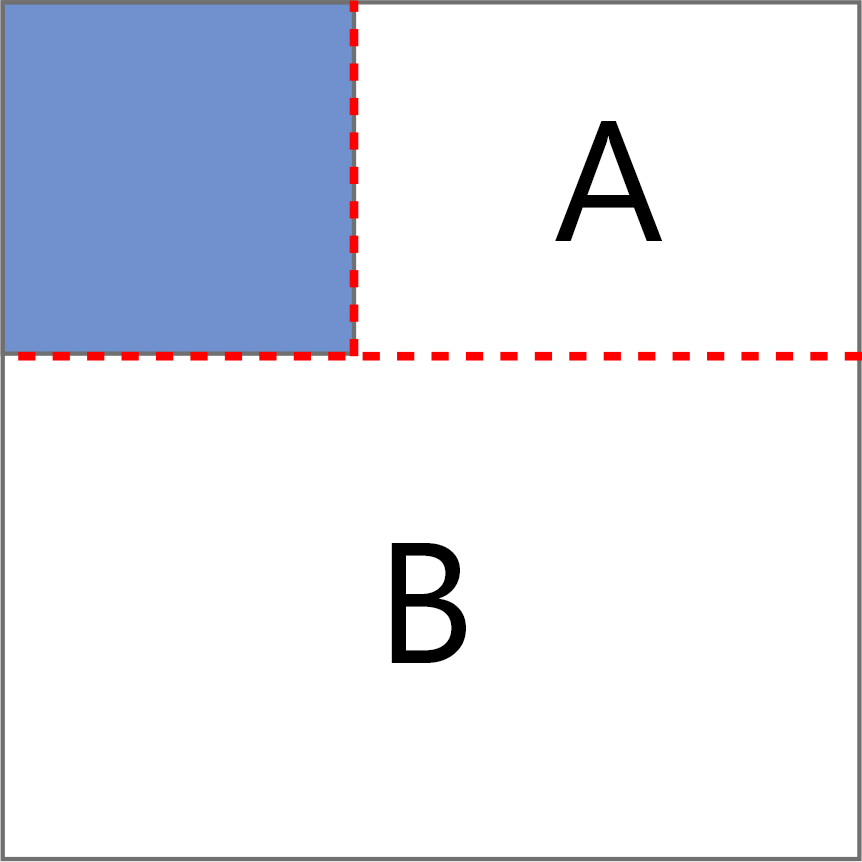
1. A continuación, se divide la plancha en 2 zonas, la zona que incluye la caja (Zona A) y la zona libre(Zona B), en la cuál se podrán almacenar más cajas.



1. Si la zona que incluye la caja no está completamente llena, esta se dividirá en otras dos zonas, la zona que es ocupada completamente por la caja y la zona libre de ella.



1. Finalmente, el tablero queda definido de la siguiente manera.



1. Para las siguientes cajas que se deseen agregar, se realizará el mismo algoritmo para cada zona libre.
2. Si una caja no se puede insertar en ninguno de los espacios libres de la plancha, se procederá a crear otra plancha y se repetirá el algoritmo.

**Complejidad**

**Tiempo**

Para cada nueva caja que se desee ingresar en la plancha, se recorrerán todas las zonas creadas. Por lo que se puede deducir que la complejidad del algoritmo es cuadrática, es decir, Big(O) = n ^ 2.

**Espacio**

Para cada nueva caja que se desee agregar a la plancha, se crearán tres zonas en la mayoría de casos, una zona con la caja y 2 zonas libres. Por lo que se deduce que la complejidad de memoria es lineal, es Big(O) = 3 \* n = n

* 1. First Fit Decreasing Height Bin Packing

**Estrategia**

Es un algoritmo heurístico constructivo que se basa en una búsqueda de fuerza bruta para poder determinar en qué planchas podrán encajar los bloques.

**Funcionamiento**

1. Se ordena descendentemente el arreglo de bloques, con respecto a la altura de cada uno.



1. Se procede a insertar el bloque más alto disponible en la plancha en caso se pueda.



1. Se define un “nivel”, un área limitada en el eje X por el ancho de la plancha y en el eje Y desde la base del bloque previamente insertado hasta la altura de este.



1. Se repite el paso 2 si el siguiente bloque a insertar pueda entrar en la plancha y su base esté al mismo nivel que la base de un “nivel” y en caso el bloque a insertar no vaya a pertenecer a un “nivel”, se continúa con el paso 3, de lo contrario se salta dicho paso.



1. Se crea una nueva plancha y se continúa con el paso 1 hasta terminar de insertar todos los bloques del arreglo.



**Complejidad**

**Tiempo**

Al inicio, se ordena descendentemente el arreglo de bloques con respecto a su altura, lo cual sería un tiempo asintótico O(n\*log(n)), donde n es la cantidad de bloques. Luego, en el peor de los casos, cada bloque a insertar creará 1 nuevo nivel, por lo que cada 1 de estos tendrá que recorrer los niveles creados anteriormente para determinar si estará en uno o se creará otro, dando resultado a un tiempo asintótico de [w (w + 1) ] / 2, donde w es la cantidad de niveles. Por lo tanto, la complejidad del algoritmo es O(w ^ 2), donde w es la cantidad de niveles.

**Espacio**

En el peor de los casos, cada bloque crearía 1 nivel, por lo tanto la complejidad del espacio utilizado sería O(w), donde w es la cantidad de niveles.

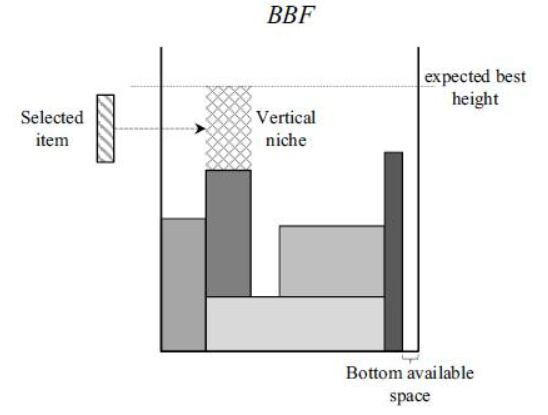
* 1. Bricklaying Best Fit Bin Packing

**Estrategia**

Es un algoritmo heurístico constructivo que se basa en la estrategia del mejor ajuste, que intenta colocar el bloque en el espacio más ajustado, con la esperanza de que esta es la mejor forma de llenarlo.

**Funcionamiento**

1. Primero se debe ordenar los bloques de manera descendente respecto a su área. Además de que el ancho de cada bloque sea mayor igual que su altura, en caso contrario se debe intercambiar los valores de las dimensiones..
2. Se crea un arreglo del tamaño del ancho del tablero para mantener los niveles de cada columna.
3. Se coloca el primer bloque disponible, el cual marcará la altura de referencia inicial igual a la altura de este.
4. Se busca la brecha de nivel más bajo y se busca el mejor bloque que pueda completarlo.Eso quiere decir el bloque que llene lo más posible esa brecha sin pasar la altura de referencia.
5. Cuando ya no se puede colocar más bloques por la altura de referencia, esta se cambia a un valor extremadamente grande para que esta no sea una limitante para colocar los bloques. Y se repite desde el paso 2.
6. En caso ya no se puedan colocar más bloques en los espacios restante de la tabla se reinicia el algoritmo con una nueva tabla con los bloques restantes.



**Complejidad**

**Tiempo**

Para cada brecha que se encuentra se busca en el arreglo de bloques disponibles el que mejor se pueda ingresar. Además cada vez que se ingresa se actualiza el arreglo de peso de columnas. Llegando a la conclusión de un algoritmo cuadrático por el tamaño de arreglo de columnas, es decir ,Big(O) = (w \* n ^ 2).

**Espacio**

Se crea una arreglo de columnas en cada tablero dando como espacios Big(O) = w

1. Experimentación

* **Tabla comparativa**

En la siguiente tabla podemos observar los resultados obtenidos al someter los tres algoritmos a cinco datasets distintos. Se están considerando el número de planchas creadas por los algoritmos, el desperdicio generado en porcentaje, el tiempo de ejecución de los algoritmos y la memoria consumida por los mismos.

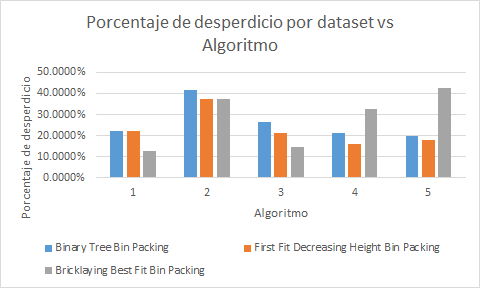
[**Datasets**](https://gist.github.com/Raigiku/0d90d7b101c5283014be8d3708bdcedd)utilizados para los algoritmos

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Binary Tree Bin Packing** | | | | |
| **Dataset** | 1 | 2 | 3 | 4 | 5 |
| **Planchas** | 1483 | 89 | 316 | 1724 | 14810 |
| **Desperdicio** | 22.2137% | 41.6798% | 26.5999% | 21.4053% | 19.8333% |
| **Tiempo de ejecución** | 1.7645 | 0.0030 | 0.0407 | 1.8641 | 153.0618 |
| **Memoria consumida** | 1557.6 | 60.8 | 268 | 1636 | 63880 |

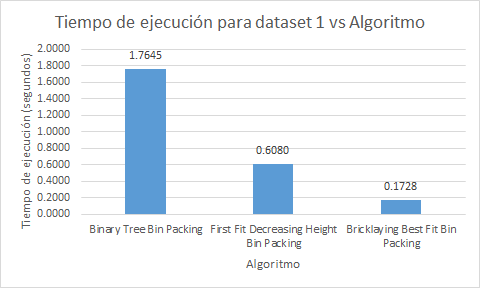
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **First Fit Decreasing Height Bin Packing** | | | | |
| **Dataset** | 1 | 2 | 3 | 4 | 5 |
| **Planchas** | 1485 | 83 | 294 | 1613 | 14441 |
| **Desperdicio** | 22.3184% | 37.4639% | 21.1074% | 15.9961% | 17.7896% |
| **Tiempo de ejecución** | 0.6080 | 0.0011 | 0.02003392 | 0.6776 | 58.9523 |
| **Memoria consumida** | 720 | 36.8 | 147.2 | 797.6 | 6920 |

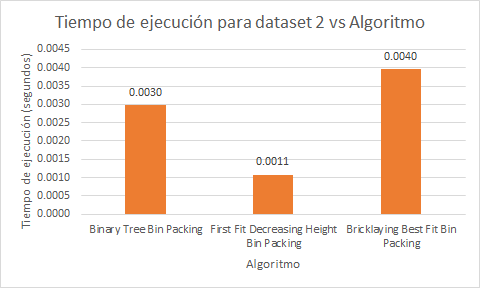
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Bricklaying Best Fit Bin Packing** | | | | |
| **Dataset** | 1 | 2 | 3 | 4 | 5 |
| **Planchas** | 1323 | 83 | 271 | 2004 | 13147 |
| **Desperdicio** | 12.8064% | 37.4639% | 14.4117% | 32.3866% | 42.3615% |
| **Tiempo de ejecución** | 0.1728 | 0.0040 | 0.0604 | 0.1841 | 26.3506 |
| **Memoria consumida** | 5010.4 | 181.6 | 2332 | 8950.4 | 209324 |

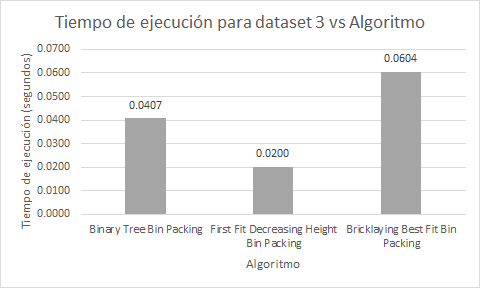
* **Porcentaje de desperdicio**

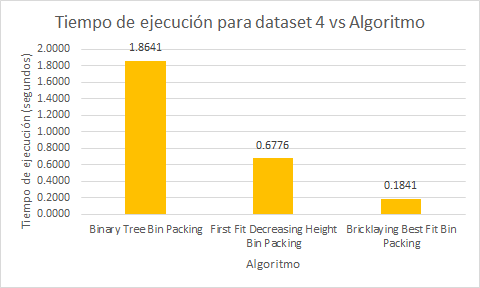
****

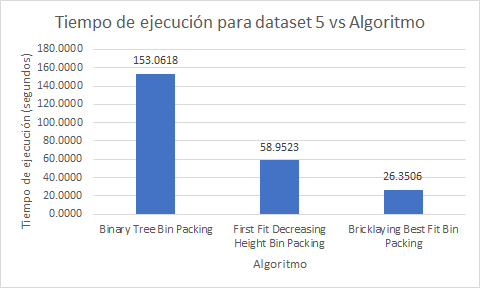
* **Tiempo de ejecución**



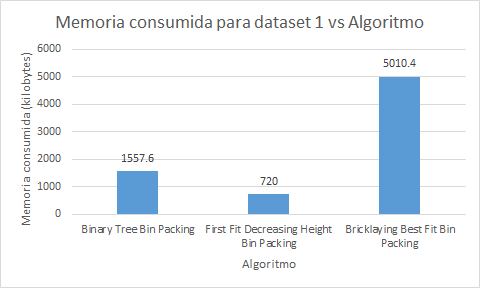


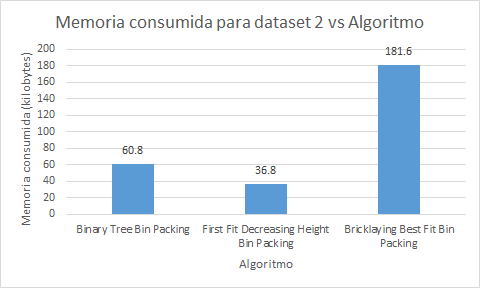


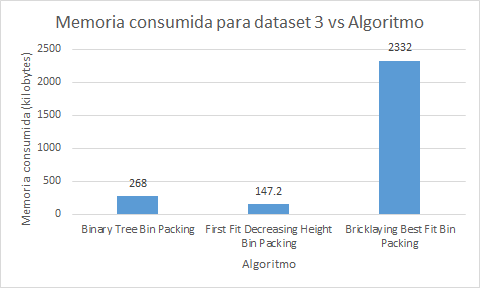


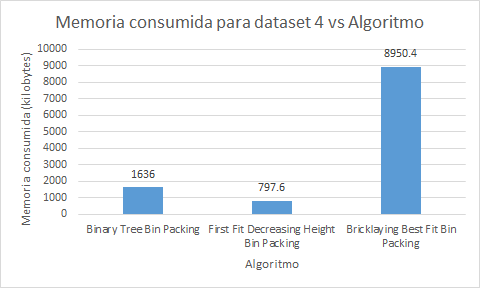


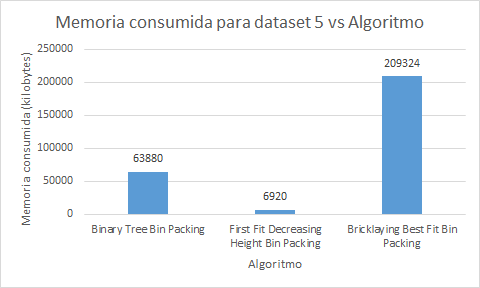
**Memoria consumida (kilobytes)**

****

****

****

****

****

1. Conclusiones

**Binary Tree Bin Packing**

* Ventajas
  + Aunque sus soluciones no son las más eficientes, el algoritmo es rápido a diferencia de otros algoritmos exactos, esto se debe a su naturaleza constructiva.
* Desventajas
  + Una de las principales desventajas del algoritmo es su consumo en memoria, ya que para cada nueva caja que se agrega, se generan tres nodos en el árbol.
* Mejoras
  + En lugar de almacenar todos los nodos del árbol, se deberían almacenar sólo los nodos que no tengan una caja asignada, con eso reducirá la complejidad de tiempo y de espacio del algoritmo

**First Fit Decreasing Height Bin Packing**

* Ventajas
  + Si todos los bloques entran en 1 solo nivel, solo se tendrá que comprobar que el siguiente bloque entre en el espacio restante, por lo que el tiempo asintótico sería O(n).
  + Debido a que al inicio del algoritmo todos los bloques son ordenados descendentemente, habrán pocos niveles desperdiciados.
* Desventajas
  + Si cada bloque debe crear 1 nivel para ser insertado, el tiempo asintótico sería O(w^2), donde w es la cantidad de niveles.
  + No se pueden apilar bloques encima de otros a menos que encima no esté dentro de un nivel.
* Mejoras
  + Durante el ordenamiento, ordenar según la dimensión del bloque más alta y en caso de esta ser el ancho, girar el bloque.
  + Permitir subniveles dentro de niveles para así permitir que se apilen más bloques.

**Bricklaying Best Fit Bin Packing**

* Ventajas
  + Debido a que es un algorítmico heurístico constructivo, esto le permite tener una solución más rápido comparado de los algoritmos exactos.
  + A comparación del BF(Best Fit), este utiliza una altura de referencia que se va actualizando lo que permite superar su desventaja que tiene respecto a los bloques que tienen una mayor altura que anchura.
* Desventajas
  + No te encuentra la solución óptima.
  + El ancho del tablero afecta directamente a la complejidad de algoritmo como del uso de memoria.
* Mejoras
  + A partir de una solución para un tablero, se debería verificar si se puede repetir. En caso sea así, se quitan los bloques y se agrega el área perdida simulando la repetición de esa solución y así se evita repetir lo mismo varias veces.

1. Referencias

* Enlace a los datasets utilizados en el proyecto

<https://gist.github.com/Raigiku/0d90d7b101c5283014be8d3708bdcedd>

* Enlace para los algoritmos

<http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0101-74382016000200197>

<https://www.researchgate.net/publication/220993640_A_Bricklaying_Best-Fit_Heuristic_Algorithm_for_the_Orthogonal_Rectangle_Packing_Problem>

http://cgi.csc.liv.ac.uk/~epa/surveyhtml.html