BoxPacker Documentation

Release version 2

Doug Wright

Sep 04, 2017

Contents

BoxPacker is an implementation of the "4D" bin packing/knapsack problem i.e. given a list of items, how many boxes do you need to fit them all in.

Especially useful for e.g. e-commerce contexts when you need to know box size/weight to calculate shipping costs, or even just want to know the right number of labels to print.

CHAPTER 1

License

BoxPacker is licensed under the MIT license.

Installation

The recommended way to install BoxPacker is to use Composer. From the command line simply execute the following to add `dvdoug/boxpacker` to your project's `composer.json` file. Composer will automatically take care of downloading the source and configuring an autoloader:

```
composer require dvdoug/boxpacker
```

If you don't want to use Composer, the code is available to download from GitHub

Requirements

BoxPacker is compatible with all versions of PHP 5.4+ (including PHP 7 and HHVM)

Versioning

BoxPacker follows Semantic Versioning. For details about differences between releases please see What's new

Principles of operation

Bin packing is an NP-hard problem and there is no way to always achieve an optimum solution without running through every single permutation. But that's OK because this implementation is designed to simulate a naive human approach to the problem rather than search for the "perfect" solution.

This is for 2 reasons:

1. It's quicker
2. It doesn't require the person actually packing the box to be given a 3D diagram explaining just how the items are supposed to fit.

At a high level, the algorithm works like this:

- Pack largest (by volume) items first
- Pack vertically up the side of the box
- Pack side-by-side where item under consideration fits alongside the previous item
- Only very small overhangs are allowed (10%) to prevent items bending in transit
- The available width/height for each layer will therefore decrease as the stack of items gets taller
- If more than 1 box is needed to accommodate all of the items, then aim for boxes of roughly equal weight (e.g. 3 medium size/weight boxes are better than 1 small light box and 2 that are large and heavy)

Getting started

BoxPacker is designed to integrate as seamlessly as possible into your existing systems, and therefore makes strong use of PHP interfaces. Applications wanting to use this library will typically already have PHP domain objects/entities representing the items needing packing, so BoxPacker attempts to take advantage of these as much as possible by allowing you to pass them directly into the Packer rather than needing you to construct library-specific datastructures first. This also makes it much easier to work with the output of the Packer - the returned list of packed items in each box will contain your own objects, not simply references to them so if you want to calculate value for insurance purposes or anything else this is easy to do.

Similarly, although it's much more uncommon to already have 'Box' objects before implementing this library, you'll typically want to implement them in an application-specific way to allow for storage/retrieval from a database. The Packer also allows you to pass in these objects directly too.

To accommodate the wide variety of possible object types, the library defines two interfaces `BoxPacker\Item` and `BoxPacker\Box` which define methods for retrieving the required dimensional data - e.g. `getWidth()`. There's a good chance you may already have at least some of these defined.

If you do happen to have methods defined with those names already, **and they are incompatible with the interface expectations**, then this will be only case where some kind of wrapper object would be needed.

Examples

Packing a set of items into a given set of box types

```
<?php
    use DVDoug\BoxPacker\Packer;
    use DVDoug\BoxPacker\Test\TestBox;  // use your own object
    use DVDoug\BoxPacker\Test\TestItem; // use your own object

    $packer = new Packer();

    /*
     * Add choices of box type - in this example the dimensions are passed in␣
↪directly via constructor,
     * but for real code you would probably pass in objects retrieved from a database␣
↪instead
```

```
     */
    $packer->addBox(new TestBox('Le petite box', 300, 300, 10, 10, 296, 296, 8,␣
↪1000));
    $packer->addBox(new TestBox('Le grande box', 3000, 3000, 100, 100, 2960, 2960, 80,
↪ 10000));

    /*
     * Add items to be packed - e.g. from shopping cart stored in user session. Again,
↪ the dimensional information
     * (and keep-flat requirement) would normally come from a DB
     */
    $packer->addItem(new TestItem('Item 1', 250, 250, 12, 200, true));
    $packer->addItem(new TestItem('Item 2', 250, 250, 12, 200, true));
    $packer->addItem(new TestItem('Item 3', 250, 250, 24, 200, false));

    $packedBoxes = $packer->pack();

    echo "These items fitted into " . count($packedBoxes) . " box(es)" . PHP_EOL;
    foreach ($packedBoxes as $packedBox) {
        $boxType = $packedBox->getBox(); // your own box object, in this case TestBox
        echo "This box is a {$boxType->getReference()}, it is {$boxType->
↪getOuterWidth()}mm wide, {$boxType->getOuterLength()}mm long and {$boxType->
↪getOuterDepth()}mm high" . PHP_EOL;
        echo "The combined weight of this box and the items inside it is {$packedBox->
↪getWeight()}g" . PHP_EOL;

        echo "The items in this box are:" . PHP_EOL;
        $itemsInTheBox = $packedBox->getItems();
        foreach ($itemsInTheBox as $item) { // your own item object, in this case␣
↪TestItem
            echo $item->getDescription() . PHP_EOL;
        }
    }
```

Does a set of items fit into a particular box

```
<?php
    /*
     * To just see if a selection of items will fit into one specific box
     */
    $box = new TestBox('Le box', 300, 300, 10, 10, 296, 296, 8, 1000);

    $items = new ItemList();
    $items->insert(new TestItem('Item 1', 297, 296, 2, 200, false));
    $items->insert(new TestItem('Item 2', 297, 296, 2, 500, false));
    $items->insert(new TestItem('Item 3', 296, 296, 4, 290, false));

    $volumePacker = new VolumePacker($box, $items);
    $packedBox = $volumePacker->pack(); //$packedBox->getItems() contains the items␣
↪that fit
```

Advanced usage

Used / remaining space

After packing it is possible to see how much physical space in each `PackedBox` is taken up with items, and how much space was unused (air). This information might be useful to determine whether it would be useful to source alternative/additional sizes of box.

At a high level, the `getVolumeUtilisation()` method exists which calculates how full the box is as a percentage of volume.

Lower-level methods are also available for examining this data in detail either using `getUsed[Width|Length|Depth()` (a hypothetical box placed around the items) or `getRemaining[Width|Length|Depth()` (the difference between the dimensions of the actual box and the hypothetical box).

Note: BoxPacker will always try to pack items into the smallest box available

Example - warning on a massively oversized box

```
<?php

    // assuming packing already took place
    foreach ($packedBoxes as $packedBox) {
        if ($packedBox->getVolumeUtilisation() < 20) {
            // box is 80% air, log a warning
        }
    }
```

Custom Constraints

For more advanced use cases where greater control over the contents of each box is required (e.g. legal limits on the number of hazardous items per box, or perhaps fragile items requiring an extra-strong outer box) you may implement the `BoxPacker\ConstrainedItem` interface which contains an additional callback method allowing you to decide whether to allow an item may be packed into a box or not.

As with all other library methods, the objects passed into this callback are your own - you have access to their full range of properties and methods to use when evaluating a constraint, not only those defined by the standard `BoxPacker\Item` interface.

Example - only allow 2 batteries per box

```
<?php
    use DVDoug\BoxPacker\Box;
    use DVDoug\BoxPacker\Item;
    use DVDoug\BoxPacker\ItemList;

    class LithiumBattery implements ConstrainedItem
    {

        /**
```

```
         * @param ItemList $alreadyPackedItems
         * @param TestBox  $box
         *
         * @return bool
         */
        public function canBePackedInBox(ItemList $alreadyPackedItems, Box $box)
        {
            $batteriesPacked = 0;
            foreach ($alreadyPackedItems as $packedItem) {
              if ($packedItem instanceof LithiumBattery) {
                  $batteriesPacked++;
              }
            }

            if ($batteriesPacked < 2) {
                return true;  // allowed to pack
            } else {
                return false; // 2 batteries already packed, no more allowed in this␣
↪box
            }
        }
    }
```

What's new / Upgrading

Note: Below is summary of important changes between versions. A full changelog, including changes in versions not yet released is available from https://github.com/dvdoug/BoxPacker/blob/master/CHANGELOG.md

Version 2

3D rotation when packing

Version 2 of BoxPacker introduces a key feature for many usecases, which is support for full 3D rotations of items. Version 1 was limited to rotating items in 2D only - effectively treating every item as "keep flat" or "ship this way up". Version 2 adds an extra method onto the `BoxPacker\Item` interface to control on a per-item level whether the item can be turned onto it's side or not.

Removal of deprecated methods

The `packIntoBox`, `packBox` and `redistributeWeight` methods were removed from the `Packer` class. If you were previously using these v1 methods, please see their implementations in https://github.com/dvdoug/BoxPacker/blob/1.x-dev/Packer.php for a guide on how to achieve the same results with v2.