BoxPacker Documentation

Release version 3

Doug Wright

Dec 02, 2023

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

ONE

LICENSE

BoxPacker is licensed under the MIT license.

1.1 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

1.1.1 Requirements

BoxPacker v3 is compatible with PHP 7.4+

1.1.2 Versioning

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

1.2 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

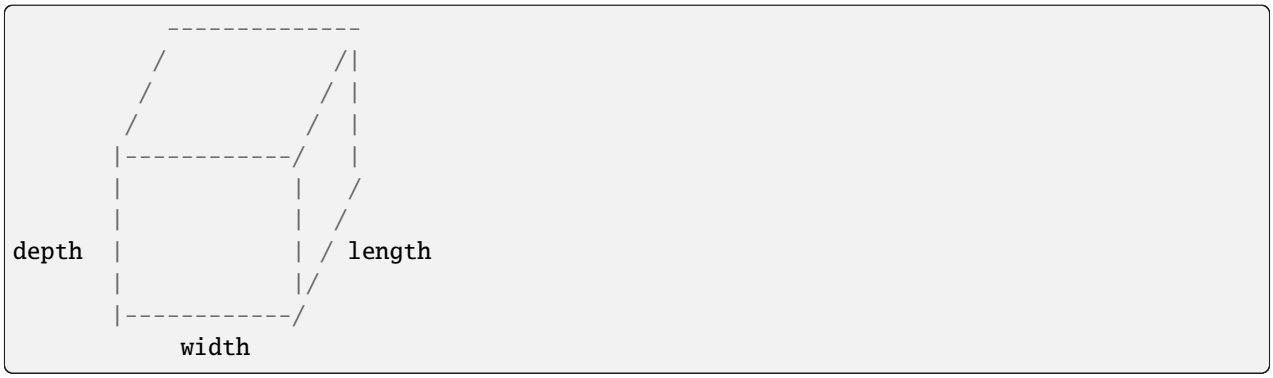
- If more than 1 box is needed to accommodate all of the items, then aim for boxes of roughly equal weight (e.g. 3 medium size/weight boxes are better than 1 small light box and 2 that are large and heavy)

1.3 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.

1.3.1 Examples

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

```
<?php
    use DVDoug\BoxPacker\Rotation;
    use DVDoug\BoxPacker\Packer;
    use DVDoug\BoxPacker\Test\TestBox;  // use your own `Box` implementation
    use DVDoug\BoxPacker\Test\TestItem; // use your own `Item` implementation

    $packer = new Packer();

    /*
     * Add choices of box type - in this example the dimensions are passed in directly␣
↪via constructor,
```

(continues on next page)

(continued from previous page)

```
     * but for real code you would probably pass in objects retrieved from a database␣
↪instead
     */
    $packer->addBox(
        new TestBox(
            reference: 'Le petite box',
            outerWidth: 300,
            outerLength: 300,
            outerDepth: 10,
            emptyWeight: 10,
            innerWidth: 296,
            innerLength: 296,
            innerDepth: 8,
            maxWeight: 1000
        )
    );
    $packer->addBox(
        new TestBox(
            reference: 'Le grande box',
            outerWidth: 3000,
            outerLength: 3000,
            outerDepth: 100,
            emptyWeight: 100,
            innerWidth: 2960,
            innerLength: 2960,
            innerDepth: 80,
            maxWeight: 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(
        item: new TestItem(
            description: 'Item 1',
            width: 250,
            length: 250,
            depth: 12,
            weight: 200,
            allowedRotation: Rotation::KeepFlat
        ),
        qty: 1
    );
    $packer->addItem(
        item: new TestItem(
            description: 'Item 2',
            width: 250,
            length: 250,
            depth: 12,
```

(continues on next page)

(continued from previous page)

```
            weight: 200,
            allowedRotation: Rotation::KeepFlat
        ),
        qty: 2
    );
    $packer->addItem(
        item: new TestItem(
            description: 'Item 3',
            width: 250,
            length: 250,
            depth: 24,
            weight: 200,
            allowedRotation: Rotation::BestFit
        ),
        qty: 1
    );

    $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;
        $packedItems = $packedBox->getItems();
        foreach ($packedItems as $packedItem) { // $packedItem->getItem() is your own␣
↪item object, in this case TestItem
            echo $packedItem->getItem()->getDescription() . PHP_EOL;
        }
    }
```

Does a set of items fit into a particular box

```
<?php
    use DVDoug\BoxPacker\Rotation;
    use DVDoug\BoxPacker\Packer;
    use DVDoug\BoxPacker\Test\TestBox;  // use your own `Box` implementation
    use DVDoug\BoxPacker\Test\TestItem; // use your own `Item` implementation

    /*
     * To just see if a selection of items will fit into one specific box
     */
    $box = new TestBox(
        reference: 'Le box',
        outerWidth: 300,
```

(continues on next page)

(continued from previous page)

```
        outerLength: 300,
        outerDepth: 10,
        emptyWeight: 10,
        innerWidth: 296,
        innerLength: 296,
        innerDepth: 8,
        maxWeight: 1000
    );

    $items = new ItemList();
    $items->insert(
        new TestItem(
            description: 'Item 1',
            width: 297,
            length: 296,
            depth: 2,
            weight: 200,
            allowedRotation: Rotation::BestFit
        )
    );
    $items->insert(
        new TestItem(
            description: 'Item 2',
            width: 297,
            length: 296,
            depth: 2,
            weight: 500,
            allowedRotation: Rotation::BestFit
        )
    );
    $items->insert(
        new TestItem(
            description: 'Item 3',
            width: 296,
            length: 296,
            depth: 4,
            weight: 290,
            allowedRotation: Rotation::BestFit
        )
    );

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

1.4 Rotation

1.4.1 Items

BoxPacker gives you full control of how (or if) an individual item may be rotated to fit into a box, controlled via the `getKeepFlat()` method on the `BoxPacker\Item` interface.

Best fit

To allow an item to be placed in any orientation.

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

    class YourItem implements Item
    {
        public function getKeepFlat(): bool
        {
            return false;
        }
    }
```

Keep flat

For items that must be shipped "flat" or "this way up".

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

    class YourItem implements Item
    {
        public function getKeepFlat(): bool
        {
            return true;
        }
    }
```

1.4.2 Boxes

BoxPacker operates internally by positioning items in "rows", firstly by placing items across the width of the box, then when there is no more space starting a new row further along the length.

However, due to the nature of the placement heuristics, better packing is sometimes achieved by going the other way i.e. placing items along the length first. By default BoxPacker handles this by trying packing both ways around, transposing widths and lengths as appropriate.

For most purposes this is fine, when the boxes come to be packed in real life it is done via the top and the direction used for placement doesn't matter. However, sometimes the "box" being given to BoxPacker is actually a truck or other side-loaded container and in these cases it is sometimes desirable to enforce the packing direction.

This can be done when using the `VolumePacker` by calling the `packAcrossWidthOnly` method.

```php
<?php
use DVDoug\BoxPacker\VolumePacker;

$volumePacker = new VolumePacker($box, $items);
$volumePacker->packAcrossWidthOnly();
$packedBox = $volumePacker->pack();
```

1.5 Sortation

BoxPacker (mostly) uses "online" algorithms, that is it packs sequentially, with no regard for what comes next. Therefore the order of items, or the order of boxes are of crucial importance in obtaining good results.

By default, BoxPacker will try to be as smart as possible about this, packing larger/heavier items into the bottom of a box, with smaller/lighter items that might get crushed placed above them. It will also prefer to use smaller boxes where possible, rather than larger ones.

However, BoxPacker also allows you to influence many of these decisions if you prefer.

1.5.1 Items

You may wish to explicitly pack heavier items before larger ones. Or larger ones before heavier ones. Or prefer to keep items of a similar "group" together (whatever that might mean for your application). The `ItemList` class supports this via two methods.

Supplying a pre-sorted list

If you already have your items in a pre-sorted array (e.g. when using a database `ORDER BY`, you can construct an `ItemList` directly from it. You can also use this mechanism if you know that all of your items have identical dimensions and therefore having BoxPacker sort them before commencing packing would just be a waste of CPU time.

```php
$itemList = ItemList::fromArray($anArrayOfItems, true); // set the 2nd param to true if␣
↪presorted
```

Overriding the default algorithm

First, create your own implementation of the `ItemSorter` interface implementing your particular requirements:

```php
/**
 * A callback to be used with usort(), implementing logic to determine which Item is a␣
↪higher priority for packing.
 */
YourApplicationItemSorter implements DVDoug\BoxPacker\ItemSorter
{
    /**
     * Return -1 if $itemA is preferred, 1 if $itemB is preferred or 0 if neither is␣
↪preferred.
     */
    public function compare(Item $itemA, Item $itemB): int
    {
```

(continues on next page)

(continued from previous page)

```
        // your logic to determine ordering goes here. Remember, that Item is your own␣
↪object,
        // and you have full access to all methods on it, not just the ones from the␣
↪Item interface
    }
}
```

Then, pass this to the `ItemList` constructor

```
$sorter = new YourApplicationItemSorter();
$itemList = new ItemList($sorter);
```

Enforcing strict ordering

Regardless of which of the above methods you use, BoxPacker's normal mode of operation is to respect the sort ordering *but not at the expense of packing density*. If an item in the list is too large to fit into a particular space, BoxPacker will temporarily skip over it and will try the next item in the list instead.

This typically works well for ecommerce, but in some applications you may want your custom sort to be absolutely determinative. You can do this by calling `beStrictAboutItemOrdering()`.

```
$packer = new Packer();
$packer->beStrictAboutItemOrdering(true); // or false to turn strict ordering off again

$volumePacker = new VolumePacker(...);
$volumePacker->beStrictAboutItemOrdering(true); // or false to turn strict ordering off␣
↪again
```

1.5.2 Box types

BoxPacker's default algorithm assumes that box size/weight is a proxy for cost and therefore seeks to use the smallest/lightest type of box possible for a set of items. However in some cases this assumption might not be true, or you may have alternate reasons for preferring to use one type of box over another. The `BoxList` class supports this kind of application-controlled sorting via two methods.

Supplying a pre-sorted list

If you already have your items in a pre-sorted array (e.g. when using a database `ORDER BY`, you can construct an `BoxList` directly from it.

```
$boxList = BoxList::fromArray($anArrayOfBoxes, true); // set the 2nd param to true if␣
↪presorted
```

Overriding the default algorithm

First, create your own implementation of the `BoxSorter` interface implementing your particular requirements:

```
/**
 * A callback to be used with usort(), implementing logic to determine which Box is
↪"better".
 */
YourApplicationBoxSorter implements DVDoug\BoxPacker\BoxSorter
{
    /**
     * Return -1 if $boxA is "best", 1 if $boxB is "best" or 0 if neither is "best".
     */
    public function compare(Box $boxA, Box $boxB): int
    {
        // your logic to determine ordering goes here. Remember, that Box is your own␣
↪object,
        // and you have full access to all methods on it, not just the ones from the Box␣
↪interface
    }
}
```

Then, pass this to the `BoxList` constructor

```
$sorter = new YourApplicationBoxSorter();
$boxList = new BoxList($sorter);
```

1.5.3 Choosing between permutations

In a scenario where even the largest box type is not large enough to contain all of the items, BoxPacker needs to decide which is the "best" possible first box, so it can then pack the remaining items into a second box (and so on). If there are two different box types that each hold the same number of items (but different items), which one should be picked? What if one of the boxes can hold an additional item, but is twice as large? Is it better to minimise the number of boxes, or their volume?

By default, BoxPacker will optimise for the largest number of items in a box, with volume acting as a tie-breaker. This can also be changed:

Overriding the default algorithm

First, create your own implementation of the `PackedBoxSorter` interface implementing your particular requirements:

```
/**
 * A callback to be used with usort(), implementing logic to determine which PackedBox␣
↪is "better".
 */
YourApplicationPackedBoxSorter implements DVDoug\BoxPacker\PackedBoxSorter
{
    /**
     * Return -1 if $boxA is "best", 1 if $boxB is "best" or 0 if neither is "best".
     */
    public function compare(PackedBox $boxA, PackedBox $boxB): int
```

(continues on next page)

(continued from previous page)

```
    {
        // your logic to determine "best" goes here
    }
}
```

Then, pass this to the `Packer`

```
$sorter = new YourApplicationPackedBoxSorter();

$packer = new Packer();
$packer->setPackedBoxSorter($sorter);
```

1.6 Weight distribution

If you are shipping a large number of items to a single customer as many businesses do, it might be that more than one box is required to accommodate all of the items. A common scenario which you'll have probably encountered when receiving your own deliveries is that the first box(es) will be absolutely full as the warehouse operative will have tried to fit in as much as possible. The last box by comparison will be virtually empty and mostly filled with protective inner packing.

There's nothing intrinsically wrong with this, but it can be a bit annoying for e.g. couriers and customers to receive e.g. a 20kg box which requires heavy lifting alongside a similarly sized box that weighs hardly anything at all. If you have to send two boxes anyway, it would be much better in such a situation to have e.g. an 11kg box and a 10kg box instead.

Happily, this smoothing out of weight is handled automatically for you by BoxPacker - once the initial dimension-only packing is completed, a second pass is made that reallocates items from heavier boxes into any lighter ones that have space.

For most use-cases the benefits are worth the extra computation time - however if a single "packing" for your scenarios involves a very large number of permutations e.g. thousands of items, you may wish to tune this behaviour.

By default, the weight distribution pass is made whenever the items fit into 12 boxes or less. To reduce (or increase) the threshold, call `setMaxBoxesToBalanceWeight()`

```
<?php
    use DVDoug\BoxPacker\Packer;

    $packer = new Packer();
    $packer->setMaxBoxesToBalanceWeight(3);
```

Note: A threshold value of either 0 or 1 will disable the weight distribution pass completely

1.7 Too-large items

As a library, by default BoxPacker makes the design choice that any errors or exceptions thrown during operation are best handled by you and your own code as the appropriate way to to handle a failure will vary from application to application. There is no attempt made to handle/recover from them internally.

This includes the case where there are no boxes large enough to pack a particular item. The normal operation of the Packer class is to throw an `NoBoxesAvailableException`. If your application has well-defined logging and monitoring it may be sufficient to simply allow the exception to bubble up to your generic handling layer and handle like any other runtime failure. Applications that do that can make an assumption that if no exceptions were thrown, then all items were successfully placed into a box.

Alternatively, you might wish to catch the exception explicitly and have domain-specific handling logic e.g.

```
<?php
    use DVDoug\BoxPacker\NoBoxesAvailableException;
    use DVDoug\BoxPacker\Packer;
    use DVDoug\BoxPacker\Test\TestBox;  // use your own `Box` implementation
    use DVDoug\BoxPacker\Test\TestItem; // use your own `Item` implementation

    try {
        $packer = new Packer();

        $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));

        $packer->addItem(new TestItem('Item 1', 2500, 2500, 20, 2000,␣
↪Rotation::BestFit));
        $packer->addItem(new TestItem('Item 2', 25000, 2500, 20, 2000,␣
↪Rotation::BestFit));
        $packer->addItem(new TestItem('Item 3', 2500, 2500, 20, 2000,␣
↪Rotation::BestFit));

        $packedBoxes = $packer->pack();
    } catch (NoBoxesAvailableException $e) {
        $problemItem = $e->getItem(); //the custom exception allows you to retrieve the␣
↪affected item
        // pause dispatch, email someone or any other handling of your choosing
    }
```

However, an `Exception` is for exceptional situations and for some businesses, some items being too large and thus requiring special handling might be considered a normal everyday situation. For these applications, having an `Exception` thrown which interrupts execution might be not be wanted or be considered problematic.

BoxPacker also supports this workflow with the `InfalliblePacker`. This class extends the base `Packer` and automatically handles any `NoBoxesAvailableException`. It can be used like this:

```
<?php
    use DVDoug\BoxPacker\InfalliblePacker;
    use DVDoug\BoxPacker\Test\TestBox;  // use your own `Box` implementation
    use DVDoug\BoxPacker\Test\TestItem; // use your own `Item` implementation
```

(continues on next page)

(continued from previous page)

```
    $packer = new InfalliblePacker();

    $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));

    $packer->addItem(new TestItem('Item 1', 2500, 2500, 20, 2000, Rotation::BestFit));
    $packer->addItem(new TestItem('Item 2', 25000, 2500, 20, 2000, Rotation::BestFit));
    $packer->addItem(new TestItem('Item 3', 2500, 2500, 20, 2000, Rotation::BestFit));

    $packedBoxes = $packer->pack(); //same as regular Packer

    // It is *very* important to check this is an empty list (or not) when exceptions␣
↪are disabled!
    $unpackedItems = $packer->getUnpackedItems();
```

1.8 Positional information

It is possible to see the precise positional and dimensional information of each item as packed. This is exposed as x,y,z co-ordinates from origin, alongside width/length/depth in the packed orientation.

```
z    y
|   /
|  /
| /
|/
0------x
```

Example:

```
<?php

    // assuming packing already took place
    foreach ($packedBoxes as $packedBox) {
        $packedItems = $packedBox->getItems();
        foreach ($packedItems as $packedItem) { // $packedItem->getItem() is your own␣
↪item object
            echo $packedItem->getItem()->getDescription() .  ' was packed at co-ordinate
↪' ;
            echo '(' . $packedItem->getX() . ', ' . $packedItem->getY() . ', ' .
↪$packedItem->getZ() . ') with ';
            echo 'l' . $packedItem->getLength() . ', w' . $packedItem->getWidth() . ', d
↪' . $packedItem->getDepth();
            echo PHP_EOL;
        }
    }
```

1.9 Limited supply boxes

In standard/basic use, BoxPacker will assume you have an adequate enough supply of each box type on hand to cover all eventualities i.e. your warehouse will be very well stocked and the concept of "running low" is not applicable.

However, if you only have limited quantities of boxes available and you have accurate stock control information, you can feed this information into BoxPacker which will then take it into account so that it won't suggest a packing which would take you into negative stock.

To do this, have your box objects implement the `BoxPacker\LimitedSupplyBox` interface which has a single additional method over the standard `BoxPacker\Box` namely `getQuantityAvailable()`. The library will automatically detect this and use the information accordingly.

1.10 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\ConstrainedPlacementItem` 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.

1.10.1 Example - only allow 2 batteries per box

```php
<?php
    use DVDoug\BoxPacker\PackedBox;

    class LithiumBattery implements ConstrainedPlacementItem
    {

        /**
         * Max 2 batteries per box.
         *
         * @param  Box            $box
         * @param  PackedItemList $alreadyPackedItems
         * @param  int            $proposedX
         * @param  int            $proposedY
         * @param  int            $proposedZ
         * @param  int            $width
         * @param  int            $length
         * @param  int            $depth
         * @return bool
         */
        public function canBePacked(
            Box $box,
            PackedItemList $alreadyPackedItems,
            int $proposedX,
            int $proposedY,
            int $proposedZ,
```

(continues on next page)

(continued from previous page)

```
            int $width,
            int $length,
            int $depth
        ): bool {
            $batteriesPacked = 0;
            foreach ($alreadyPackedItems as $packedItem) {
              if ($packedItem->getItem() instanceof LithiumBattery) {
                  $batteriesPacked++;
              }
            }

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

1.10.2 Example - don't allow batteries to be stacked

```
<?php
    use DVDoug\BoxPacker\PackedBox;
    use DVDoug\BoxPacker\PackedItem;

    class LithiumBattery implements ConstrainedPlacementItem
    {

        /**
         * Batteries cannot be stacked on top of each other.
         *
         * @param  Box            $box
         * @param  PackedItemList $alreadyPackedItems
         * @param  int            $proposedX
         * @param  int            $proposedY
         * @param  int            $proposedZ
         * @param  int            $width
         * @param  int            $length
         * @param  int            $depth
         * @return bool
         */
        public function canBePacked(
            Box $box,
            PackedItemList $alreadyPackedItems,
            int $proposedX,
            int $proposedY,
            int $proposedZ,
            int $width,
            int $length,
            int $depth
```

(continues on next page)

(continued from previous page)

```
        ): bool {
            $alreadyPackedType = array_filter(
                iterator_to_array($alreadyPackedItems, false),
                function (PackedItem $item) {
                    return $item->getItem()->getDescription() === 'Battery';
                }
            );

            /** @var PackedItem $alreadyPacked */
            foreach ($alreadyPackedType as $alreadyPacked) {
                if (
                    $alreadyPacked->getZ() + $alreadyPacked->getDepth() === $proposedZ &&
                    $proposedX >= $alreadyPacked->getX() && $proposedX <= (
↪$alreadyPacked->getX() + $alreadyPacked->getWidth()) &&
                    $proposedY >= $alreadyPacked->getY() && $proposedY <= (
↪$alreadyPacked->getY() + $alreadyPacked->getLength())) {
                    return false;
                }
            }

            return true;
        }
    }
```

1.11 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 try to pack items into the smallest box available

---

1.11.1 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
        }
    }
```

1.12 What's new / Upgrading

Note: Below is summary of key changes between versions that you should be aware of. A full changelog, including changes in minor versions is available from https://github.com/dvdoug/BoxPacker/blob/master/CHANGELOG.md

1.12.1 Version 3

Positional information on packed items

Version 3 allows you to see the positional and dimensional information of each item as packed. Exposing this additional data unfortunately means an API change - specifically `PackedBox->getItems` now returns a set of `PackedItem` s rather than `Item` s. A `PackedItem` is a wrapper around around an *Item* with positional and dimensional information (x/y/z co-ordinates of corner closest to origin, width/length/depth as packed). Adapting existing v2 code to v3 is simple:

Before

```
<?php
    $itemsInTheBox = $packedBox->getItems();
    foreach ($itemsInTheBox as $item) { // your own item object
        echo $item->getDescription() . PHP_EOL;
    }
```

After

```
<?php
    $packedItems = $packedBox->getItems();
    foreach ($packedItems as $packedItem) { // $packedItem->getItem() is your own item␣
↪object
        echo $packedItem->getItem()->getDescription() . PHP_EOL;
    }
```

If you use `BoxPacker\ConstrainedItem`, you'll need to make the same change there too.

PHP 7 type declarations

Version 3 also takes advantage of the API break opportunity introduced by the additional positional information and is the first version of BoxPacker to take advantage of PHP7's type declaration system. The core `BoxPacker\Item` and `BoxPacker\Box` interfaces definitions have been supplemented with code-level type information to enforce expectations. This is a technical break only, no implementation requires changing - only the correct type information added, e.g.

Before

```
<?php
    /**
     * @return string
     */
    public function getDescription()
    {
        return $this->description;
    }
```

After

```
<?php
    /**
     * @return string
     */
    public function getDescription(): string
    {
        return $this->description;
    }
```

1.12.2 Version 2

3D rotation when packing

Version 2 of BoxPacker introduces a key feature for many use-cases, 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/src/Packer.php for a guide on how to achieve the same results with v2.