Packing rectangles into rectangular bins for best coverage

Contents

[Introduction 3](#_Toc514667650)

[Problem statement 3](#_Toc514667651)

[Measuring success 3](#_Toc514667652)

[Subdividing stock 3](#_Toc514667653)

[Packing strategies and algorithms 4](#_Toc514667654)

[Iteration and Recursion 4](#_Toc514667655)

[Brute force packing 4](#_Toc514667656)

[Combinations strategy 5](#_Toc514667657)

[Best combination per board strategy 5](#_Toc514667658)

[Multi-threading 6](#_Toc514667659)

# Introduction

Woodworkers routinely face the problem of how to cut parts from stock in such an arrangement that they waste the least amount of wood. In this article we will explore a strategy developed and implemented in a computer application to plan how to accomplish this.

# Problem statement

In its simplest form, this problem can be stated as this:

How can a set of rectangular shapes (called “parts”) be packed into a second set of bigger rectangular shapes (called “boards”) without overlapping or rotating, to minimize the uncovered area (waste)?

# Measuring success

Success would be measured using the following criteria

* Was each large rectangle optimally covered?
* Was the process completed in a practical timeframe?
* Is the output simple enough, yet detailed enough to allow it to be used as a cutting plan?
* Do the pieces get arranged in a practical arrangement, (easy to saw through)

I will take you through my journey of developing a strategy and eventually a program to perform this cut list planning process.

# Subdividing stock

When placing a part on a board, the remaining unoccupied area of the board can be represented by two smaller rectangles in either a horizontal or vertical arrangement as illustrated below.

Vertical Horizontal

Part

V1

V2

Part

H1

H2

These new partial boards can be used for placing further parts. This strategy of subdividing the remainder of the board follows closely the way in which a woodworker would saw a board of wood and allows the cut plan to be applied easily to the original use case without impractical sawing being required.

# Packing strategies and algorithms

There are many so-called packing algorithms for packing a set of shapes into bins. Each with its own advantages and disadvantages. Complexity, effectivity and time required are among the measurements by which these are evaluated.

Usually these algorithms don’t consider the practicalities of sawing a wooden board and as a result they sometimes produce packings with internal corners that are not easily transferred into the workshop. The design of this algorithm will be centred around a method if subdivision that will allow for easy sawing patterns.

# Iteration and Recursion

In computer languages there are two methods for stepping through a set of items. These methods are called iteration and recursion.

Iteration is the process of stepping through the list in a linear fashion from one end to the other and is usually used on items in a linear data structure like an array or list. These are implemented using so-called “For”-loops, or similar constructs.

Recursion is the process where a set of instructions (called a function) makes calls to itself. This technique is much more difficult to implement but allows us to solve some very complex problems where the underlying data or solution is hierarchical using relatively little code.

The process of packing a board with parts can be described like this:

Remove a part from the list of required parts and place it on the board.

Then, pack the remainder (the uncovered area) of the board with the remaining required parts.

This type of process where the solution is expressed as a function of a subset of the original set of input arguments is usually solved easier using recursive implementations.

# The strategy / algorithm

As the application was being developed, it became clear how difficult it is to implement a solution to this problem in software, even though the human brain performs the task with ease. The algorithm designed and implemented uses both iteration and recursion to accomplish the process using a small amount of code.

Originally a brute force strategy was tried:

* Iterate through the available parts
  + Find the smallest bin that will accommodate the current part
    - If none is found, the packing fails
  + If all parts are packed
    - If this packing wastes less than previous packings
      * Remember this packing and discard previous best packing
  + If there are parts left to pack
    - Subdivide the bin into its remaining pieces and remove the piece occupied by the part
      * Attempt to pack the remaining parts onto the remaining bins using this same algorithm (recursion)

Although this would produce good results, the rime required would soon become impractical. The number of different combinations/packings to be evaluated by the algorithm (N) in terms of the number of parts(n) and a single board can be express follows:

Brute force packing is a strategy used to pack blocks into bins. In short this algorithm follows these steps:

* Iterate through all the blocks
  + Place the current block into the bin
    - If all blocks placed we are done and can exit
    - If all the bocks are not placed, but the bin is full we failed and need to return one level to try the next block
    - If there are blocks remaining,
      * Pack the remaining blocks into the remaining bin area by calling the same procedure
  + If the rest of the block could not be placed, place the next block

This is called the brute force method because no intelligence is used to pick which block to place or where. It simply steps blindly through each sequence of placement to hopefully get to a solution. This strategy will find the best solution, but it may take hours or even days to complete.

# Combinations strategy

This strategy first generates a list of possible combinations for the blocks and then tests the combinations for fit.

The problem here is the sheer amount of combinations that need to be generated and tested.

For a collection of “n” items, the number of unique combinations in which these items can be arranged (“N”), can be calculated as

Using this formula, we calculate the following examples:

|  |  |
| --- | --- |
| **Number of items** | **Number of unique combinations** |
| 1 | 1 |
| 2 | 3 |
| 3 | 7 |
| 4 | 15 |

But if we had merely 50 items, the number of unique combinations explodes to a staggering 1,125,899,906,842,623.

For this reason, we must also filter out any combinations we are not interested in and we do so using the cumulative area of the combination. Any combination of blocks with a cumulative area larger than the area of the bin will never be able to fit into the bin and can be discarded.

The list of combinations is sorted in descending order according to their cumulative area. Then, starting at the first combination with the biggest cumulative area, we iterate through these combinations and attempt to pack the parts into the bin. The first combination that can successfully be packed into the bin would be the one resulting in the least remaining bin area (waste) and would therefore be considered the most efficient.

Discarding the combinations with too big cumulative area still leaves thousands of combinations to test though. For this reason, we also discard the combinations with very low cumulative areas. These combinations will not produce favourable fittings anyway.

# Best combination per board strategy

In this strategy, we calculate the combinations of blocks for each bin separately where the cumulative area of the combination is between 90% and 100% of the area of the bin.

The combinations are sorted in descending order of the cumulative area and each is then tested using a brute force algorithm to check if all the members of the combination can be packed into the bin.

This results in a “best combination” for each bin given the available blocks. The best amongst these (the one with the least waste) is stored and, and is set aside. The other solutions and combinations is discarded.

The process is then repeated with the remaining blocks and remaining bins in the same manner until no bins are left or no blocks are left to pack in them.

# Multi-threading

If this process was implemented in a normal single threaded application, it will still be impractical as it would take very long to converge to a solution. Luckily I was able to implement the application in a multi-threaded manner where the combinations for boards are generated and tested for all the boards at the same time. This reduced the required time dramatically, and the application is now able to produce satisfactory cut plans in practical timeframes of a few minutes.

New packing strategy/algorithm:

Start every board with a point top left and bottom right.

Bottom right point disabled.

Loop through enabled points for board…points sorted by Y

For each, determine the maximum length and width of it’s potential placement area by looking at the other points to its right/bottom.

Find the biggest part that will fit in the area.

OR….find the biggest combination of parts that can fit in a row or column in the area.

Place the part and disable the point it was placed on/at.

Create two new points – top right and bottom left of the new placing, offsetting them for saw kerf and insert to retain sorting on Y asc.

We need to distinguish a few different scenarios here:

If the new placement caused a bottom left point further down from another, but on the same X, disable that upper point.

We need to draw out the different scenarios for placings and make sure we cover them when creating the 2 points.