Two-dimensional Stock Cutting and Rectangle Packing:
Binary Tree Model Representation for Local Search
Optimization Methods

Vassilios S. Vassiliadis*

Department of Chemical Engineering,
University of Cambridge,
Pembroke Street, Cambridge CB2 3RA,
United Kingdom

6 October 2003

Abstract

This paper presents a binary tree hierarchical representation for bounding boxes (rectangles) comprised of shapes (other rectangles in our case) that are to be cut from a two-dimensional sheet of material. Although tree-representations of this problem have been presented in the open literature extensively, the binary-tree representation is shown in this work to be capable of capturing any configuration of such rectangular shapes in 2-D space, such as rotations through right angles and translations, and to be advantageous and more general over other approaches when local search optimization procedures are to be utilized. The construction is such that these boxes must be bound together in pairs having a common edge, and can be extended to contain constrains regarding vertical or horizontal space between the actual objects as well as special types of cuts, e.g. guillotine cuts used in the glass cutting industry. Finally, a simple continuous sheet application is shown as a demonstration of the capabilities of the algorithm in connection with a local search method, specifically threshold accepting.

Keywords

rectangle packing, two-dimensional knapsack problem, trim loss minimization, local optimization search methods, simulated annealing, threshold accepting, stock cutting

---

* E-mail: vsv20@cheng.cam.ac.uk
1. Introduction and literature survey

This paper presents a methodology suitable for local search methods when these are applied to two-dimensional space partitioning problems. Such problems include the trim loss minimization problem, which finds many applications, as for example in the cutting of 2-D shapes from wood, marble, glass, fabric, etc. In these applications the loss of material due to the placement and cutting constraints is the issue of concern and its minimization is the goal of related optimization procedures.

The problem of may be addressed through mathematical programming applications, but it involves formulations of NP-hardness, hence can be applied to only problems involving a small number of items to be placed on the raw 2-D sheet for cutting. Things may become more complicated when considering a multitude of constraints, as for example the placement of an array of objects on different available stocks with the aim the maximization of economy of material, also ensuring the reusability of discarded, unused parts. A search in the open literature reveals a number of approaches to the 2D trim loss minimization problem (also equivalently to 2D rectangle packing). Such approaches may be classified as below.

Branch and bound

The mixed integer-linear programming approach (MILP) is an approach which is based mainly on formulating the problem through a mathematical programming statement and solving it by branch-and-bound (B&B) or cutting-plane search methods. It is in a sense closed, as it allows proof of global optimality of the solutions if the B&B method is run to full exhaustion of the combinatorial tree, subject to node fathoming. However, for large instances the problem is still difficult to solve routinely to global optimality due to its NP-hard complexity. The interested reader is referred to an excellent paper outlining a branch-and-bound approach and special properties for the optimal packing of rectangles by Scheithauer (1997), and to the work of Hadjiconstantinou and Christofides (1995) on two-dimensional Knapsack problems.
Heuristics

Heuristics are based on either intuitive, experience-based rules as to how to place and cut objects, or are supplemented by graph-theoretic algorithms that at least try to optimize to some extent the placement of objects. For example, Scheithauer and Sommerweiß (1996) use the 4-block heuristic for the rectangle packing problem (RPP).

Fritsch and Vornberger (1995) consider the problem of two-dimensional cutting stock arising in wood, metal and the glass industry. Their method is based on an iterative algorithm heuristically, based on maximum weight matching of rectangles into metarectangles handled via a graph-theoretical way, also utilising shape functions so that the orientations of the demand rectangles remain free (not fixed) until the complete layout has been computed.

Local search methods

Local search methods rely on techniques such as Simulated Annealing, Threshold Accepting, and may be considered to include also to a certain extent Genetic Algorithms, in attempting to mimic physical or biological processes in order to arrive to a better configuration than the starting one (or more than one in case of population-based methods such as genetic algorithms).

Beasley (2000) introduces a nonlinear mathematical formulation for two-dimensional non-guillotine cutting with a population heuristic as a solution technique (closely related to genetic algorithms). Corno et al. (1997) also introduce the use of a genetic algorithm for the minimization of area loss in flat glass cutting, taking into account many industrial practice constraints particular to glass-cutting, and demonstrate a very good computational performance of their technique. Interestingly, they stress a motivating factor for developing rapid solution procedures, namely that “a requirement is that the optimization must run in real-time, concurrently with the cutting operation”.

- 3 -
2. Local search methods: an overview

Although the above sited literature is not an exhaustive attempt to account for the enormous bulk of work appearing on the theme topic of this paper, nonetheless it is considered as a representative cross section of advances and ideas in the field as it currently stands. Motivated by the work sited, and from continuous bibliographical search, we observe that interestingly there is still a missing ingredient in terms of an adequate representation of the two-dimensional rectangle packing (or stock cutting problem) suitable for local search methods, such as simulated annealing (SA) (Kirkpatrick et al., 1983) and threshold accepting (TA) (Dueck and Scheuer, 1990).

Without going into the details of the SA and TA algorithms, which are well-established and reviewed in the OR community, it is mentioned here that they rely on being able to define a structure of the optimization problems involved such that one solution evolves into another one by minor modifications of the decision parameters. In other words, a datastructure representation of any solution needs to be available and decisions on degrees of freedom (e.g. in the RPP, the placement of the rectangles) will be able to modify the solution locally so that a new solution is obtained by this manipulation.

Both the SA and TA methods define such moves modifying one solution into another as local neighbourhood perturbations. Both methods also allow solutions to yield a higher objective function value over the previous one and still be accepted as the current working solution. This strategy is what makes them capable of allowing “climbing” out of (escaping) local minima towards regions where the objective function is much improved. Although by far they are not guaranteed global optimization methods, such as B&B, practice has proven them to be very effective tools for the rapid improvement of solutions for combinatorially hard discrete optimization problems.

Finally, these methods are also called metaheuristics as they translate the problem of optimizing any underlying problem into finding the most suitable settings (driver
parameters) of the algorithm (cooling scheme for SA, and threshold reduction scheme for TA).

We next proceed into a detailed description of a suitable model encapsulation for the RRP and trim loss minimization (two-dimensional stock cutting problem) in the next section.
3. The binary tree datastructure for rectangle placement and cutting

The placement of rectangles within a larger piece of cutting stock material (slabs) is considered in this section. Effectively, what will follow is an algorithmic description and elaboration as to how this is achieved using a binary tree, and how various operational constraints can be encapsulated into the moves associated with local search algorithms, such as SA and TA outlined in the previous section.

3.1 Stock slabs and orders

Here we outline assumptions for the case where the slabs of material have finite height and width, while we cater also for the rectangles to be packed into vertical cut traverses that have to be cut with guillotine cuts (which is true for many materials, such as glass). Also, if a traverse as a whole does not fit, it will be transferred to a new slab. These are assumptions used to define a description of algorithmic steps and problem modelling, and are by no means restrictive.

1. Assume \( h_{\text{slab}} \) and \( w_{\text{slab}} \) (height and width respectively, vertical and horizontal)
2. Assume \( h_{\text{slab}} \) and \( w_{\text{slab}} \) are hard constraints that cannot be violated during searches
3. Assume that if piece arrangement is over-spilling in width (x-direction), a new slab will be used to accommodate the traverses that do this
4. In width overspill: use a new slab, each time allocating part of the tree structure to a new number of slabs according to which traverses move out
5. A traverse cannot have width more than that of the slab, and accordingly calculate separately:
   a. \% filling of 1\textsuperscript{st} slab
   b. \% filling of all slabs needed
   c. decide on which objective to use later

The orders are assumed to be sets of same rectangles to be delivered to the customers; the problem is depicted schematically in Figure 1.
3.1 Arrangement of rectangles on binary tree to form a slab of material

Each slab is represented as a binary tree. The details of this problem encapsulation are:

1. Assume there are NP pieces to cut out of the slabs
2. Pieces will be arranged strictly feasibly in terms of $h_{slab}$
3. Each piece will be the leaf at the last layer of a pairing binary tree
4. Total layers: $NL = \text{round up} (\ln_2 NP)$, such that the last layer in the binary tree has $2^{NL} \geq NP$ pieces.
5. As the last layer has more or equal leaves to NP some of the leaves will be allocated the empty box; there will be precisely $NE = 2^{NL} - NP$ leaves.
6. The total tree containing the NP pieces contains $NNTOT = 2^{NL+1} - 1$ nodes.

This is not so dramatic for even up to 1,000 pieces to be cut.

The logic is thus that all required glass pieces (orders) are placed at once on a total placement binary tree. The algorithm will decide which pieces and how they are fit into the first slab, and possibly if based on one type of stock, how they are fit to a number of slabs. Multiple stocks can be handled in a similar manner.

For example, for 7 pieces $NL = 3$. Thus 7 pieces will be placed at the last layer and 1 piece is the “E” empty piece. This is shown in Figure 2.

Consider that we are pairing the rectangles two at a time. The node properties defining the bounding box of the pairing of rectangles A and B are:

1. Rotation of A
2. Rotation of B
3. Size of A
4. Size of B
5. Cut orientation of box AB

The pairing of two nodes (or super nodes) is indicated in Figure 3.
In terms of orientation of pieces (nodes and super nodes) the following can be observed:

- Rotation of total object:
  - Variable $y_{\text{rot}}^{\text{node}} = \{1, \text{yes OR. 0, no}\}$
  - Theoretically this can be applied to all nodes

- Cut orientation
  - Variable $y_{\text{cut}}^{\text{node}} = \{1, \text{vertical OR. 0, horizontal}\}$
  - Applied to all nodes, excluding leaves at the bottom (last layer) of the binary tree (meaningless to apply there)

The composition rules, from individual pieces to the whole slab coverage, using the binary tree representation are as follows:

1. Mother node $AB = \text{daughter A + daughter B}$
   a. Height $h_{AB}$ and width $w_{AB}$
2. Pieces A and B oriented in AB according to cut orientation
   a. $y_{AB}^{\text{cut}} = 1$ (vertical) (Figure 4a)
      - Tallest piece assumed to be located at left (for final analysis this does not make a difference anyway)
      - $h_{AB} = \max \{h_A, h_B\}$
      - $w_{AB} = w_A + w_B$
   b. $y_{AB}^{\text{cut}} = 0$ (horizontal) (Figure 4b)
      - Longest piece assumed to be located at bottom (for final analysis this does not make a difference anyway)
      - $w_{AB} = \max \{w_A, w_B\}$
      - $h_{AB} = h_A + h_B$
3. Covers also the case of A or B or both being empty boxes (h=0, w=0)

Finally, the properties of the daughters A and B combined to yield the mother AB node are as follows:
$y_{\text{node}}^{\text{rot}} = 1$ (rotated) OR $0$ (un-rotated), referring to rotating the original orientation input by the user, or to leave it as is provided initially (un-rotated). For rectangles A and B this yields:

Node A:
- $y_{A}^{\text{rot}} = 0$
  - $h_{A} = h_{A}^{0}$, $w_{A} = w_{A}^{0}$
- $y_{A}^{\text{rot}} = 1$
  - $w_{A} = h_{A}^{0}$, $h_{A} = w_{A}^{0}$

Node B:
- $y_{B}^{\text{rot}} = 0$
  - $h_{B} = h_{B}^{0}$, $w_{B} = w_{B}^{0}$
- $y_{B}^{\text{rot}} = 1$
  - $w_{B} = h_{B}^{0}$, $h_{B} = w_{B}^{0}$

It is necessary to introduce “empty” nodes as many as required in the pairing stages to allow an object (rectangle) to be “on its own”. This, during the search phase, may form later a part of a triplet of an object already containing 2 or more other objects. In this fashion, this allows the freedom of not to couple only things in pairs but also in triplets.

At present it can be only conjectured how many empty nodes one would need, but perhaps it makes sense that the lower bound be set at 50% of the number of actual rectangles required to be placed in slabs. This has been used as a heuristic in case studies ran with the algorithm outlined in this work. For example consider the case of 4 objects, 3 of which fit precisely on top of the largest piece. If no empties are included in the structure, then the following results as shown in Figure 5. Figure 6 shows the effect inclusion of an appropriate number of empties has on the compaction that can be achieved for the same illustrative case.

If there is a minimum piece size cutting limitation say $D$, a separation is forced then as $d$ (equal in both directions for example), given by:
If $y_{\text{node \ cut}} = 0$, horizontal cut

- If $\min \{h_A, h_B\} < D$ then
  - $h_{\text{node}} = h_A + h_B + d$
- Else
  - $h_{\text{node}} = h_A + h_B$
- Endif
- $w_{\text{node}} = \max \{w_A, w_B\}$

Else If $y_{\text{node \ cut}} = 1$, vertical cut

- If $\min \{w_A, w_B\} < D$ then
  - $w_{\text{node}} = w_A + w_B + d$
- Else
  - $w_{\text{node}} = w_A + w_B$
- Endif
- $h_{\text{node}} = \max \{h_A, h_B\}$
Endif

It is also possible to extend the formulation presented above to cater for forced spacing between rectangles placed on a slab. Different space provisions can be made for the horizontal and vertical directions as shown below.

If $y_{AB \ cut} = 1$ (vertical)

- Smallest piece assumed to be located at left (for final analysis this does not make a difference anyway)
- $h_{AB} = \max \{h_A, h_B\}$
- $w_{AB} = w_A + w_B + d_w$

Else If $y_{AB \ cut} = 0$ (horizontal)

- Longest piece assumed to be located at bottom (for final analysis this does not make a difference anyway)
- $w_{AB} = \max \{w_A, w_B\}$
- $h_{AB} = h_A + h_B + d_h$
Endif
where $d_w$ and $d_h$ are the desired separation spaces for the horizontal and vertical directions respectively.
3.2 Traverse representation and identification

Another important issue in placement for trim loss minimization are the particular constraints regarding traverses. Glass, among other materials, has to be cut by guillotine cuts, i.e. by cuts extending all the way in one direction (either the whole width or height of a slab of glass).

Locating traverses (identifying them) within a slab binary tree representation requires the following steps:

- Starting from the top node of the tree, bisect the tree in two sides, left and right
- If the top node is of orientation horizontal in its “packing” then from the whole slab an entire traverse is made.
- If the top node is of orientation vertical in its packing then the node is not a simple traverse and we have to go deeper (look at its daughter nodes).
- Do this recursively
- First encounter at any branch of the binary tree with a horizontal packing must terminate there signifying a traverse.

Figure 7 presents some symbols useful for the drawing of binary trees for rectangle packing. Figure 8 explains graphically how traverses are found and how their constraints are implemented during operation of any algorithm. Figures 9, 10 and 11 give examples of locating traverses in a binary tree, finding 4, 3 and 4 traverses in the respective cases drawn.
3.3 Implementation within a local search stochastic optimization algorithm

The stochastic search method (such as simulated annealing or threshold accepting) can be implemented according to the following algorithmic steps.

Initialization

1. Select layer NL (last layer, containing only leaves without daughters, bottom of the tree)
   a. Put all pieces 1 … NP starting at node 1 of layer NL, only using rotation if height of piece exceeds \( h_{\text{slab}} \), until all pieces are done. Pad the empty places (if any) with the empty box.
   b. Compute tree properties by composition
      - For layer = NL-1 to 0
         - For node = 1 to \( 2^{\text{layer}} \)
            - assign a vertical cut: \( y_{\text{node}}^{\text{cut}} = 1 \)
            - compute \( h_{\text{node}} = f(\text{daughter A, daughter B, } y_{\text{node}}^{\text{cut}}) \)
            - compute \( w_{\text{node}} = f(\text{daughter A, daughter B, } y_{\text{node}}^{\text{cut}}) \)
         - Next node
      - Next layer
   c. Compute objective function(s)
Local “neighborhood” perturbations

There are 3 types of feasible perturbations allowable within the scope of the previously outlined structure for the binary tree:

1. Select node for flipping of rotation:
   - Select layer randomly
     - Layer = 0..NL
   - Select node randomly
     - Node = 1..2^{Layer}
   - Flip rotation
     - $y_{\text{node}}^{\text{rot}} : 0 \rightarrow 1 \text{ or } 1 \rightarrow 0$
   - Re-compute resulting values upward parent to parent till root (Figure 4).
     - The cost of this computation is $O(\log_2 NP)$, which is the property that makes the binary tree representation most suitable for fast application of local search methods.

2. Select node for flipping of cut orientation:
   - Select layer randomly
     - Layer = 0..(NL-1)
   - Select node randomly
     - Node = 1..2^{Layer}
   - Flip cut orientation
     - $y_{\text{node}}^{\text{cut}} : 0 \rightarrow 1 \text{ or } 1 \rightarrow 0$
   - Re-compute resulting values upward parent to parent till root (as in Figure 12)
3. Select two nodes in the same layer (level) of the binary tree to exchange them

   - Select layer randomly
     - Layer = 1..NL

   - Select 2 nodes randomly
     - Node1, Node2 = 1..2\(^{\text{Layer}}\); Node1 different to Node2.
     - Node1 and Node 2 must not have the same parent node as their exchange will yield the same result.

   - Exchange nodes

   - Re-compute resulting values upward parent to parent till root (as in Figure 12 before), for both the perturbed (exchanged) locations (Figure 13).

**Acceptances**

The general criteria to accept a new layout configuration subject to a perturbation as described above are the following:

   - Height of slab is not violated by move

   .AND.

   - Traverses all fit exactly on slabs (where traverses are required by the problem formulation)

   .AND.

   - Acceptance criterion of the stochastic method (based on whatever objective we choose) is satisfied.
General algorithm and implementation issues:

1. Slab partitions end-to-end (at traverses) to be found at initialization and after every move to be updated and checked.

2. Definition of which traverses belong to 1st slab, and then count how many slabs are needed to fit the remaining traverses onto.

3. Feasibility check augmentation: traverses cannot be bigger than slab width

4. Find fast way to compute and re-compute objective function(s)
   a. Efficiency of packing first slab alone can be one possible objective function
   b. Efficiency of packing all slabs can be another objective to target, except the last one as end-effects are not to be counted: new orders will force by that time a new optimization anyway – the implementations are to be designed also for real-time usage

5. Post-solution analysis and software package evolution
   a. Find how vertical cuts (traverses of the whole slab) are to be found for the entire slab height, thus define traverses
   b. Visualization tool for solutions and perhaps evolution stages to monitor convergence path
   c. Order database management tool
3.4 Multi-slab assignment problems

Single and multiple problems may be considered. Single slab, allowing up to any number of objects to be allocated, the rest staying in the “object store”, and multiple allowing any number of objects to be allocated, the rest staying in the object store.

Figure 14 shows the single slab case and Figure 15 the multiple slab case in relation to a local search optimization algorithm with exchange moves.
4. Computational demonstration

The binary tree datastructure was implemented within a threshold accepting standard algorithm in C++. The runs were conducted on a Pentium III 750 MHz computer, using Bloodshed DevCpp, a C++ integrated development environment, interfacing GNU C++ on a Windows NT operating system. CPU times are reported in the text within a range, since to obtain a good solution stochastic search methods have to be run from different starting points.

Two main case studies are presented, focusing on a finite height and infinite width slab problem. The objective is to achieve the greatest compaction possible for a set of objects. The initial setup layout follows a greedy type filling, stacking as many objects possible with horizontal matching until the maximum height of the slab is reached, at which point a new traverse effectively starts by vertical matching.

Perturbations also result in engaging the same fill-up procedure and automatic matching orientation assignment to all nodes belonging to higher levels than the last one in the binary tree.

Case A

This case involves 7 rectangles, the sizes of which are given in Table 1 in arbitrary dimensional units. The initial layout and information are given in Figure 16. The slab height is fixed at 200 units, the bounding box width is 103 units and the bounding box height is 181 units. The final solution is depicted in Figure 17, having a bounding box of width 30 units and height 199 units. This solution is obtained in 5-10 seconds for different starting point runs.

Case A.1

This is the same problem as that of case A, with a special constraint of 5 distance units to be imposed between all rectangles. Figure 18 shows the initial arrangement
and Figure 19 the best solution obtained. The initial arrangement has a bounding box of 209 by 105 (width by height) units, and the best found a bounding box of 40 by 198 dimensional units. Again the CPU time ranged from 5 to 10 seconds.

**Case B**

This case emulates the placement of multiple orders, by requiring the optimal placement of 126 objects, obtained by 18 lots of the same 7 objects of case A. The slab height remains the same, fixed at 200 units. The initial arrangement is shown in Figure 20, with a bounding box of 677 by 200 units. The best arrangement found is presented in Figure 21, with a bounding box of 405 by 200 units. The CPU time ranged from 3 to 5 minutes.

**Case B.1**

This is the same problem as that of case 5, with a special constraint of 2 distance units to be imposed between all rectangles. Figure 22 shows the best solution obtained, with a bounding box of 510 by 200 units. The initial layout had a bounding box of 735 by 198 units. The CPU time ranged from 3 to 5 minutes.
5. Conclusion

This paper has presented a general purpose binary tree datastructure for the representation of rectangle layout problems. This representation has been demonstrated to be able to encapsulate both requirements for cost (area) minimization and compaction, but also to be able to capture other operational constraints, such as traverses of objects placed on stock material slabs.

Computational results have been also presented, not so much as to fine-tune a specific optimization procedure, but to demonstrate the applicability of the datastructure. Beyond the problems presented in this paper, several others have been run, with situations involving 1,000 and 10,000 objects. Loading time (initialization of tree in memory) was rather slow (approximately 2-3 minutes) for the latter case. However, following this, the run speed for this problem (10,000 objects) was such that it was estimated to allow about 100 million perturbations (rotations of basic rectangles and position swaps) per 40 minutes of CPU time on a 750 MHz Pentium III.

Future work is expected to be directed in tuning a suitable optimization method such as threshold accepting used in this work, and its application to cases of industrial importance.

Acknowledgements

The author would like to thank Mr. Jose Ignacio Marin Alberdi for the many useful discussions during the development of this work.
References


FIGURES

Figure 1. Example of rectangle orders and slab material stock

<table>
<thead>
<tr>
<th>Layer</th>
<th>Nodes in layer</th>
<th>Nodes in rooted subtree</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>L</td>
<td>$2^L$</td>
<td>$2^{L+1} - 1$</td>
</tr>
</tbody>
</table>

Figure 2. NL = 7 example of binary tree representation
Figure 3. Pairing of nodes into super nodes

Figure 4a. Vertical cut orientation  Figure 4b. Horizontal cut orientation

Figure 5. The case of 4 objects not compacting to the maximum possible by pair wise coupling
Figure 6. The case of 4 objects *compacting* to the maximum possible by pair wise coupling

A horizontally packed node, potentially it is a traverse signifier (its bounding box defines it) if it is encountered at a higher level in the tree.

A vertically packed node, it cannot possibly be a traverse signifier (its bounding is not stacked on top, it can be split in further verticals possibly).

Figure 7. Drawing symbols to distinguish horizontal and vertical pairing of rectangles
Figure 8. Rectangle orientation, relationship with cuts and traverses (where applicable)

The first node (root) is horizontal, hence it indicates that one vertical cut will release the objects from the slab.

By default, all simple pieces (bottom layer) are set to be horizontally "cut" and they must not be perturbed from this state (meaningless)

Figure 9. Example of traverse identification through binary tree representation
Figure 10. Example of traverse identification through binary tree representation

Figure 11. Example of traverse identification through binary tree representation
Figure 12. Re-computation from bottom layer node to top node (root) bounding box

Figure 13. Sub-tree exchanges (at same level always) and associated re-computations of root node bounding box
Figure 14. Single slab and piece placement by exchange moves from “store”

Figure 15. Multiple slabs and piece placement by exchange moves from “store”
Figure 16. Initial arrangement for Case A.
Figure 17. Best arrangement found for Case A.
Figure 18. Initial layout for Case A.1
Figure 19. Best layout found for Case A.1
Figure 20. Initial layout for Case B
Figure 21. Best layout found for Case B
Figure 22. Best layout found for Case B.1
TABLES

<table>
<thead>
<tr>
<th>Rectangle number</th>
<th>Width (arbitrary units)</th>
<th>Height (arbitrary units)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>103</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 1. 7 rectangle lot for computational case studies