Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2014, Article ID 959453, 12 pages
http://dx.doi.org/10.1155/2014/959453

Research Article

Mathematical Modeling and Optimal Blank Generation in Glass Manufacturing

Raymond Phillips,[1] Matthew Woolway,[1] Dario Fanucchi,[1] and M. Montaz Ali[1,2]

[1] *School of Computational and Applied Mathematics, University of the Witwatersrand, 1 Jan Smuts Avenue, Private Bag 03, WITS 2050, Johannesburg, South Africa*
[2] *TCSE, Faculty of Engineering and the Build Environment, University of the Witwatersrand, 1 Jan Smuts Avenue, Private Bag 03, WITS 2050, Johannesburg, South Africa*

Correspondence should be addressed to M. Montaz Ali; montaz.ali@wits.ac.za

Received 6 January 2014; Accepted 20 January 2014; Published 27 April 2014

Academic Editor: Aderemi Oluyinka Adewumi

This paper discusses the stock size selection problem (Chambers and Dyson, 1976), which is of relevance in the float glass industry. Given a fixed integer N, generally between 2 and 6 (but potentially larger), we find the N best sizes for intermediate stock from which to cut a roster of orders. An objective function is formulated with the purpose of minimizing wastage, and the problem is phrased as a combinatorial optimization problem involving the selection of columns of a cost matrix. Some bounds and heuristics are developed, and two exact algorithms (depth-first search and branch-and-bound) are applied to the problem, as well as one approximate algorithm (NOMAD). It is found that wastage reduces dramatically as N increases, but this trend becomes less pronounced for larger values of N (beyond 6 or 7). For typical values of N, branch-and-bound is able to find the exact solution within a reasonable amount of time.

1. Introduction

The cutting and packing of stock are important problems in the metal, paper, wood, and glass industries (amongst others). Consequently, many researchers have considered these problems as mathematical optimization problems and derived good algorithms towards their solutions. In particular, the *Stock-Cutting problem* is concerned with the cutting of specific rectangles (orders) desired by customers from larger shapes (blanks) produced during the manufacturing process. This problem was first treated as a linear programming problem in [1] for one-dimensional stock-cutting and in [2] for two-dimensional stock cutting and has since been extensively studied in various forms. Indeed, Sweeney and Paternoster [3] reviewed more than 400 books, articles, dissertations, and working papers on stock cutting and packing in 1992, and since then new work has appeared (e.g., [4, 5]). In general, the stock cutting problem is concerned with the cutting out of many smaller rectangles (or other shapes) from a fixed larger rectangle. A related, but less well-known, problem is the selection of stock sizes or blanks (the larger rectangle) from which to cut orders. This problem is of particular importance in the float glass industry, where "holding good stock sizes appears to have at least as big an impact on trim loss as cutting up the stock plates efficiently" [6].

A typical glass manufacturing plant receives hundreds of different sized orders per year for a single material and thickness of glass. A single order size will typically need to be cut hundreds, thousands, or tens of thousands of times to satisfy customer demand. In the production of float glass, a continuous ribbon of flat glass is produced in the manufacturing plant. This ribbon is cut on-line into large sizes (blanks) that are stored and cut as needed into specific order sizes. This two-stage cutting process is carried out for various practical reasons: it is costly and sometimes impossible to cut the many different order sizes directly on the float-line, and it is also sometimes infeasible to store the many different order sizes in advance. Given expected order sizes and numbers, the *stock size selection problem* is the problem of deciding which large sizes (blanks) to cut on the float line in order to minimize wastage after all the orders have been cut from these blanks.

In this paper, we study the stock size selection problem as it applies to a local South African float glass manufacturing plant. Given a list of orders and a small positive integer, N, as well as cutting limitations on the float line, we show how to find the N blank sizes to cut on the float-line that minimize the wastage when the orders are later cut from these sizes. Our treatment of the problem differs from that of [6] because we restrict our attention to the case (relevant to the industry) where only *one* type of order is cut from a blank at a time. We reduce the optimization problem to a column selection problem, which we then solve directly with depth-first search (DFS) and branch-and-bound methods and heuristically using NOMAD.

Section 2 provides a formal description of the problem which is mathematically modeled in Section 3. Section 4 introduces various algorithms for solving the problem as formulated, with results following in Section 5. Finally, some concluding remarks and insights are mentioned in Section 6.

2. Problem Description

2.1. The Meta Problem. A float glass factory receives a roster of orders from numerous clients. These orders come in a wide variety of magnitudes, shapes, and sizes. For example, motor corporations may have large orders for windscreens, while there may be some clients that require a unique, single order.

Blanks are large pieces of glass that are cut directly on the production line at the glass factory. It is from these blanks that individual orders are later cut. It is desirable to cut a small, fixed number N of different blank types on the production line (with each blank type being cut for an arbitrarily large number of times). Each order is then assigned to be cut from one of these blank types, and some glass is lost as wastage in this process. Given a value for N (typically between 2 and 6) and a list of orders, the *stock size selection problem* asks for the dimensions (widths and lengths) of the N blank types to be cut on-line so as to minimize the wastage when the orders are cut from these blanks. The dimensions are constrained by the float-line parameters and cutting machinery and thus must lie in a range between a known fixed minimum width (W_{min}) and length (L_{min}) and a known fixed maximum width (W_{max}) and length (L_{max}).

2.2. Blank Cutting and Problem Assumptions. We make three key assumptions about the process of cutting orders from blanks in this paper.

(1) Each individual blank is only ever cut out into copies of a single order, as illustrated in Figure 2 (as opposed to a complex stock-cutting problem like those considered in [2]).

(2) Each order will be cut out from only one *type* of blank. That is, each blank type can be a cutting medium for many orders, but each order can only be assigned to one blank type for cutting. Figure 1 portrays an example of this mapping relationship.

(3) The blank cannot be rotated before cutting the order.

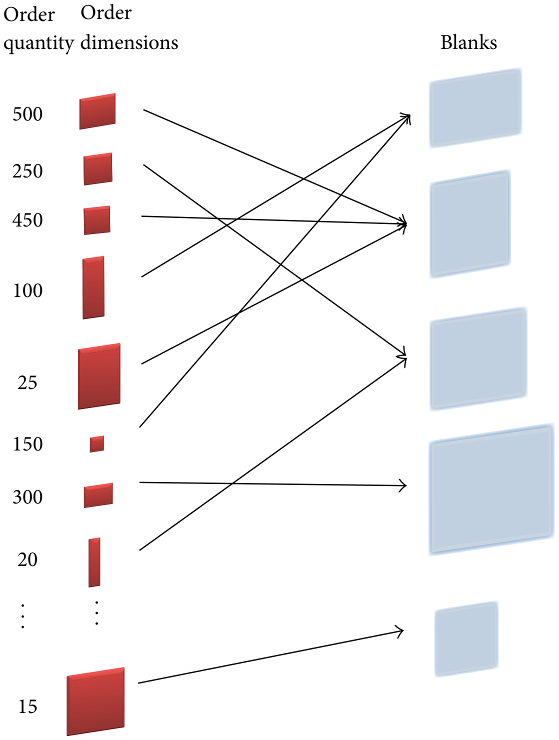

FIGURE 1: The mapping relationship of many orders to one blank.

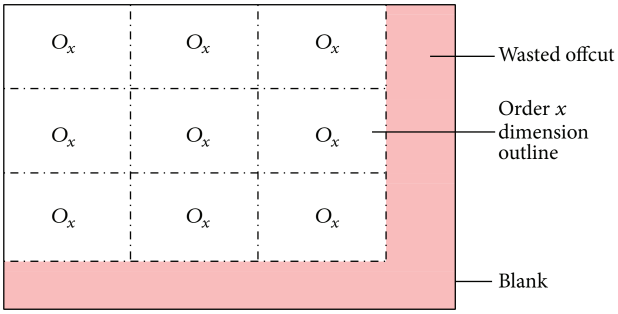

FIGURE 2: The partitioning of a blank with dimensions of an order x.

The last point above is relevant in the industry, because for certain types of flat glass it is necessary to preserve the direction of the grain relative to the order dimensions.

Referring to Figure 2, we notice a shaded region as a result of the cutting of an order x from the blank. This region represents a wasted section of the blank. Blank cutting invariably results in an off-cut wastage primarily due to the 3 cardinal problem assumptions outlined earlier. More often than not, the chosen order dimensions will not allow for "perfect fits." Indeed, Figure 2 reveals that no more of order x can be cut in the blank. However, there are cases where there is no wastage. For instance, looking at Figure 1, the last order has no off-cut since it shares the same dimensions as its blank. The primary aim of this optimization problem is to minimize the wastage from cutting by finding a set number of optimal blank types to satisfy the list of orders.

3. Model Formulation

3.1. A Basic Objective Function. Let there be M orders, each specified by a triple (n_k, w_k, l_k), where n_k indicates the quantity of the order to be cut and w_k and l_k give the width and length of the order, respectively. Let the chosen blank types have widths $W_1, W_2, \ldots, W_N$ and lengths $L_1, L_2, \ldots, L_N$. For each order index k running from 1 to M, assign it to be cut from blank $b_k \in \{1, 2, \ldots, N\}, \forall k$. The wastage associated with such a set-up is

$$f(L_1, \ldots, L_N, W_1, \ldots, W_N, b_1, \ldots, b_M) = \sum_{k=1}^{M} \left(\left\lceil \frac{n_k}{\lambda_{bk}\omega_{bk}} \right\rceil W_{b_k} L_{b_k} - n_k w_k l_k \right), \tag{1}$$

where

$$\lambda_{bk} = \left\lfloor \frac{L_{b_k}}{l_k} \right\rfloor, \qquad \omega_{bk} = \left\lfloor \frac{W_{b_k}}{w_k} \right\rfloor. \tag{2}$$

The first term in the sum indicates the quantity of blank b_k needed to cater for the kth order multiplied by the area of this blank. The second term is the total area of the kth order that needs to be cut. The difference gives the wastage on the kth order.

The above function must be minimized over all possible blank types as well as assignments of orders to blanks for cutting. The latter values are in fact uniquely determined by the former. Notice how each term in the sum that makes up the objective function is independent of the others. The choice of b_k will only influence the kth term in the sum, and therefore we can independently pick each one in such a way as to minimize this kth term. Since for each value of k there are only N values of b_k to choose from and N is typically very small, this is not a very challenging subproblem. The objective function thus only depends on the choice of blank sizes, as depicted as follows:

$$f(L_1, L_2, \ldots, L_N, W_1, W_2, \ldots, W_N) = \sum_{k=1}^{M} \min_{b_k \in \{1,2,\ldots,N\}} \left(\left\lceil \frac{n_k}{\lambda_{k,b_k}\omega_{k,b_k}} \right\rceil W_{b_k} L_{b_k} - n_k w_k l_k \right) \tag{3}$$

with λ_{k,b_k} and ω_{k,b_k} given by

$$\lambda_{k,b_k} = \left\lfloor \frac{L_{b_k}}{l_k} \right\rfloor, \qquad \omega_{k,b_k} = \left\lfloor \frac{W_{b_k}}{w_k} \right\rfloor. \tag{4}$$

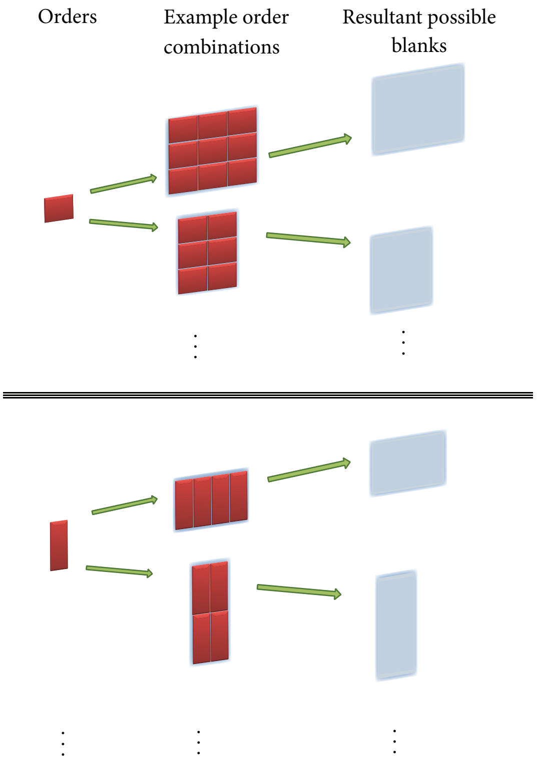

Figure 3: Using orders to create possible blanks.

3.2. Transition to Combinatorial Optimization. Before considering methods to solve the problem, we give some thought as to the search space, that is, the space of all possible dimensions of all the blanks. At first glance, this seems like a $2N$-dimensional continuous space, with two variables (width and length) associated with each blank type. This is a very bleak prospect as the objective function itself is far from continuous. It is clear, however, that very small changes in the dimensions of a blank type, b_k, will often have little qualitative effect on the solution: the same number of each order will still fit in the blank, and so ω_{bk} and λ_{bk} will be unchanged. Small decreases in L_{b_k} and W_{b_k} of this type are then guaranteed to decrease the objective function (which is locally bilinear in these regions where ω_{bk} and λ_{bk} are constant). We can thus continue to decrease L_{b_k} and W_{b_k} until L_{b_k} is a multiple of some l_k and W_{b_k} is a (possibly different) multiple of w_k. It follows that the dimensions of the blanks can be restricted to the set of multiples of the dimensions of the orders to be cut from them:

$$\begin{aligned} L_{b_k} \in \mathscr{L} &= \{C_1 l_k \mid k = 1, 2, \ldots, M; C_1 \in \mathbb{N}\} \cap [L_{\min}, L_{\max}], \\ W_{b_k} \in \mathscr{W} &= \{C_2 w_k \mid k = 1, 2, \ldots, M; C_2 \in \mathbb{N}\} \\ &\quad \cap [W_{\min}, W_{\max}], \end{aligned} \tag{5}$$

and so the search space is

$$\mathscr{B} = \mathscr{L} \times \mathscr{W}. \tag{6}$$

This set is discrete and also bounded and can easily be enumerated.

Figure 3 details two examples on how orders' dimensions are utilized to generate the search space of possible blanks.

The problem is now fundamentally a combinatorial optimization problem. The objective is to optimally choose a finite, predefined number (N) of blanks from $\mathscr{B}$ and then optimally assign orders to be cut in these respective choices, graphically seen in Figure 1, such that wastage is minimised.

3.3. The Cost Matrix Data Structure. It is possible to restructure the optimization problem introduced above into a very suggestive form that is very simple to write down and reason about. To achieve this, a 2-dimensional array, called the *cost matrix*, is employed, each row representing an order and each column a blank from the creation procedure outlined in the previous section. Each element (k, b_k) in this matrix represents the waste associated with cutting order k from blank type b_k, where

$$c_{k,b_k} = \left(\left\lceil \frac{n_k}{\lambda_{k,b_k}\omega_{k,b_k}} \right\rceil W_{b_k} L_{b_k} - n_k w_k l_k \right) \tag{7}$$

with λ_{k,b_k} and ω_{k,b_k} as defined in (4).

We can write the matrix $c = (c_{ij})$, where $i \in \{1, \ldots, M\}$ and $j \in \{1, \ldots, N\}$. For example, if the element in position $(2, 3)$ of the matrix c was 12, it would indicate that cutting order 2 from blank 3 would result in a wastage of 12 units.

A point in the search space corresponds to a selection of a set S of N columns of this matrix, and the objective function can then clearly be rewritten in terms of the matrix as

$$f(S) = \sum_{i=1}^{M} \min_{j \in S} c_{ij}. \tag{8}$$

3.4. Similarity to the p-Median Problem. Interestingly, this problem is similar to another combinatorial problem known as the *p-median problem*. Briefly, the p-median problem involves locating facilities to satisfy demand points. Assigning a demand point to be satisfied by a particular facility incurs some or other costs. An example of this quantified cost may be a community having to travel a distance D kilometres to reach a designated clinic in rural Western Australia [7].

The objective of the p-median problem is to minimise the sum of these costs accrued from satisfying demand points.

Like the glass cutting problem, which stores waste values in its cost matrix, the p-median problem captures these costs in a 2D array as well. Investigating the p-median problem's integer programming formulation gives one a sense of what the cutting problem's formulation may look like [8]:

$$\min \sum_i \sum_j d_{ij} x_{ij} \tag{9}$$

subject to

$$\sum_i x_{ij} = 1, \quad \forall j, \tag{10}$$

$$x_{ij} \leq y_i, \quad \forall i, j, \tag{11}$$

$$\sum_i y_i = p, \tag{12}$$

$$x_{ij}, y_i \in \{0, 1\}, \quad \forall i, j, \tag{13}$$

$$x_{ij} = \begin{cases} 1, & \text{if demand point } j \text{ is satisfied by facility at } i, \\ 0, & \text{otherwise.} \end{cases}$$
$$y_i = \begin{cases} 1, & \text{if a facility is located at } i, \\ 0, & \text{otherwise.} \end{cases} \tag{14}$$

Notice that d_{ij} is the cost or distance associated with demand point j being satisfied by potential facility i. It is then d that is the cost matrix in the above formulation.

Constraint (10) expresses the need that all demand points must be satisfied. Constraint (11) prevents any user's demand from being satisfied from a location with no available facility. The total number of facilities, designated p, is set by constraint (12).

The glass cutting problem in this paper shares many concepts with the p-median problem.

Referring to Table 1, there are a number of noteworthy similarities. Firstly, both are minimization problems. The objective is to minimise the costs/wastage that will inherently be accumulated by how we decide to satisfy demands (p-median problem) or assign orders (glass cutting problem). These decisions can be captured in a binary decision variable like x_{ij} seen in the p-median problem formulation. Recall that only a finite number of blanks can be chosen. Moreover, recall that the p-median problem can only locate p facilities. These are analogous constraints that can be similarly formulated.

3.5. An Integer Programming Formulation. Recognizing its likeness to the p-median problem, the integer programming formulation for the glass cutting problem can be defined as follows:

$$\min \sum_i \sum_j c_{ij} x_{ij} \tag{15}$$

such that

$$\sum_j x_{ij} = 1, \quad \forall i, \tag{16}$$

$$\sum_j \theta\left(\sum_i x_{ij} \right) = N, \tag{17}$$

$$x_{ij} \in \{0, 1\}, \quad \forall i, j, \tag{18}$$

$$\theta(x) = \begin{cases} 0, & x = 0, \\ 1, & x > 0, \end{cases} \tag{19}$$

$$x_{ij} = \begin{cases} 1, & \text{if blank } j \text{ is used to cut order } i, \\ 0, & \text{otherwise.} \end{cases} \tag{20}$$

Constraint (18) defines the binary decision variable, explained above. Constraint (16) stipulates that every order must be cut. Constraint (17) limits one to only selecting N blanks to cut the orders. Equation (19) defines θ as a Heaviside function, imperative to the operation of constraint (17).

Table 1: Concept Comparison of the p-median Problem to Glass Cutting Problem.

Concept	p-median	Glass Cutting
Objective	Minimise cost; see (9)	Minimise Wastage
Task	Satisfy demand points with located facilities	Assign orders to be cut from generated blanks
Finite constraint	Can locate p facilities, see Constraint 10	Can choose N blanks
Hard constraint	Every demand must be satisfied, see Constraint 8	Every order must be satisfied
Integrality constraint	A demand point j is satisfied by one facility i. Implication: employ binary decision variable x_{ij}	An order is cut within one blank and only one blank

Table 2: Graphical representation of cost matrix c and selection process.

	B_1	$B_2{}^*$	B_3	$B_4{}^*$	$B_5{}^*$	B_6	B_7	Result
o_1	5	10^{**}	3	14	12	6	13	10^{**}
o_2	4	3^{**}	12	11	8	5	10	3^{**}
o_3	2	9^{**}	7	11	14	6	2	9^{**}
o_4	1	12	5	20	11^{**}	3	7	11^{**}
o_5	7	11	10	2^{**}	6	9	3	2^{**}
$\sum$ wastage								35^{**}

*Refers to the blanks selected and **the minimum coefficients in the selected columns.

4. Methodology

It may be prudent to visualise the forthcoming optimization procedure of the glass cutting problem. Selection and assignment ensue with the cost matrix. Consider the following cost matrix. This instance has 5 orders, seen by the number of rows, and 7 blanks, seen by the number of columns. Consider

$$c = \begin{bmatrix} 5 & 10 & 3 & 14 & 12 & 6 & 13 \\ 4 & 3 & 12 & 11 & 8 & 5 & 10 \\ 2 & 9 & 7 & 11 & 14 & 6 & 2 \\ 1 & 12 & 5 & 20 & 11 & 3 & 7 \\ 7 & 11 & 10 & 2 & 6 & 9 & 3 \end{bmatrix}. \qquad (21)$$

Looking at this matrix, the mathematical problem can be defined as follows.

Considering a matrix c, select N columns of c so that the sum of the minimum coefficients within the columns of the selected rows is as small as possible.

Table 2 shows an example array that captures wastage according to the example cost matrix c. Take note that only 3 of the 7 blanks can be chosen. In this case, blanks B_2, B_4, and B_5 have been chosen. In accordance with the definition above, the minimum coefficients of the rows in these chosen columns are selected and summed. This sum represents the total wastage from the proposed solution. It is this value that needs to be minimised.

Clearly, the choice of the columns (i.e., blanks) determines the objective function value of the problem. The assignment component is to some extent automatically performed once the columns have been chosen since the minimum coefficient of each row is chosen within the columns. The next section will explain the 3 optimization techniques utilised to effectively choose the columns.

4.1. Depth-First Search. Depth-first search (DFS) is an algorithm that explores a tree or graph data structure. The search begins at the root node of the tree, usually resembling the starting state of a problem. Its strategy is to constantly seek to branch "deeper" from the current node. If the current node has no unexplored edges, the algorithm "backtracks" to the current node's predecessor. The algorithm will attempt to branch again in the depth-first manner. This process of backtracking and branching continues until all nodes reachable from the root node have been explored [9].

It is important to note that DFS will *always* visit every state in the search space. It is guaranteed, therefore, to find the optimal solution, but if the search space is large, it can take a very long time to do so. With regard to our problem, the algorithm iterates through every possible combination of selected columns in the cost matrix. This means that the algorithm evaluates every possible combination of blanks, calculating the wastage for each of these solutions.

The root node of the tree is the empty set: no columns were selected. The children of a node are obtained by appending to the set a single new column to the right of all the columns already in the set. The leaf nodes consist of sets of size N. Reaching a leaf node in the search tree is indicative of a feasible proposed solution. The objective function is evaluated for the solution and the current is best updated if necessary. The depth-first search method we utilise is outlined in Algorithm 1.

4.2. Branch-and-Bound Method. As previously mentioned, depth-first search needs to visit every point in the search space in order to find the optimal solution. This quickly becomes prohibitive as the search space gets large. Branch-and-bound method is a modified version of depth-first search method that takes advantage of known bounds on the objective function value in order to prune the search tree down to a more manageable size. The method was first described in [10].

The basic idea behind branch-and-bound method is to keep track of an upper bound, U, on the global optimum value as the search proceeds (usually just the best value found so far) and at each node to compute a *node specific* lower-bound L_{node} on the objective function value that could be obtained by continuing to expand the children of that node. If for some node $L_{\text{node}} > U$, then that node can be *pruned* from the search without any risk of missing out on the minimum value. Branch-and-bound method is thus an exact algorithm that finds the global optimum solution, but with good bounds it can do so in a fraction of the time it would take to apply DFS.

```
Initialize cost matrix, $C$
Initialize number of blanks to select, $N$
**function** DFS($C, N, cur$)
    Set $b$ as the number of columns in $C$.
    **if** $cur$ not defined **then**
        $cur \leftarrow \emptyset$
        $last \leftarrow 0$
    **else**
        Set $last$ as the last element in the current path, $cur$
    **end if**
    **if** $N = 0$ **then**
        $path \leftarrow cur$
        $cost \leftarrow f\,(path)$
    **else**
        // Recursively “bubble” DFS to descendants
        $nchildren \leftarrow b - N - last + 1$
        Initialize ($nchildren \times 1$) array $cp$ to store cost
        values of descendants
        Initialize ($nchildren \times N + length(cur)$) array
        $paths$ to store the respective paths of descendants
        **for** $i$ from 1 to $nchildren$ **do**
            $(cp[i], paths[i]) \leftarrow$
                    DFS($C, N - 1, cur \cup \{last + i\}$)
        **end for**
        // Find best path at this junction and return it
        $cost \leftarrow \min(cp)$
        $pos \leftarrow \underset{j}{argmin} cp$
        $path \leftarrow paths[pos, :]$
        **return** $(cost, path)$
    **end if**
**end function**
```

Algorithm 1: Pseudocode for Recursive Depth-First Search.

We describe below how we constructed our upper bounds and lower bounds for branch- and-bound method, as well as some useful preprocessing.

4.2.1. Upper Bounds. Any feasible point provides an upper bound on the minimization problem. To begin with, we made use of two heuristics to generate “good” feasible points that could be used as initial upper bounds for the branch-and-bound algorithm. The first heuristic begins with all the columns included in S. It then successively removes columns greedily in such a way as to increase the cost function as little as possible on each step until there are only N columns remaining. It returns these columns as its solution. The second heuristic starts with $S = \emptyset$ and successively adds columns greedily in such a way as to decrease the cost function as much as possible on each step until there are N columns in S. It returns these columns as its solution. Empirically this tends to be a rather good guess.

As the algorithm progresses and finds new solutions, it keeps a constant record of the best solution found so far. This best solution acts as the upper bound when deciding whether to prune a node.

Before proceeding we point out a mathematical convenience that we will use when discussing the lower bounds. When at a specific node in the DFS tree we have already selected a set S_0 with N_0 columns and we will select the remaining columns (call them S_1, with $|S_1| = N - N_0$) from the part of the matrix to the right of all the columns in S_0 as we proceed with our search, we create the vector v as follows:

$$v_i = \min_{j \in S_0} c_{ij}, \quad i = 1, 2, \ldots, M. \tag{22}$$

The objective function for the remaining columns can be written as

$$f\,(S_1) = \sum_{k=1}^{M} \min\left(v_k, \min_{j \in S_1} c_{kj}\right). \tag{23}$$

This was a useful technique in the code itself because only the vector v needed to be passed down the tree in order to be able to reconstruct the full function evaluation at all of the children. It was also useful in reasoning about lower bounds, as illustrated below.

4.2.2. First Lower Bound. Let us assume that the algorithm is at a node in the tree with S_0 defined as above, and let the last column in S_0 be column r of the matrix. Then, for any choice of S_1,

$$\begin{aligned} f(S_1) &= \sum_{k=1}^{M} \min\left(v_k, \min_{j\in S_1} c_{kj}\right) \\ &\geq \sum_{k=1}^{M} \min\left(v_k, \min_{j>r} c_{kj}\right), \end{aligned} \tag{24}$$

where the inequality follows from the fact that S_1 is a subset of the set of columns with $j > r$.

This is the first lower bound: the cost at any node is bounded below by the cost of including *all remaining columns* from here onwards.

4.2.3. Second Lower Bound. Next, we ask what the *reduction* in cost would be due to the addition of some column, $a > r$, into the set. We define

$$R(a) = \sum_{k=1}^{M} \left(v_k - \min\left(v_k, c_{ka}\right)\right). \tag{25}$$

We also define the reduction in cost due to the addition of a set of columns, T:

$$R(T) = \sum_{k=1}^{M} \left(v_k - \min\left(v_k, \min_{j\in T} c_{kj}\right)\right). \tag{26}$$

Now, for two columns,

$$\begin{aligned} R(a) + R(b) &= \sum_{k=1}^{M} \{(v_k - \min(v_k, c_{ka})) \\ &\quad + (v_k - \min(v_k, c_{kb}))\} \\ &\geq \sum_{k=1}^{M} (v_k - \min(v_k, c_{ka}, c_{kb})), \end{aligned} \tag{27}$$

where the inequality follows from the fact that one of the two terms in the first sum is precisely equal to the term in the second sum, and the other is positive or zero.

This could be generalized to the result

$$\sum_{i\in T} R(i) \geq R(T). \tag{28}$$

Now, let Q be the set with $|Q| = N - N_0$ that maximizes $R(Q)$; then, clearly Q is also the set that minimizes the objective function and so its function value is the theoretical lower bound at the current vertex. Now sort the remaining columns by their R-values, pick the $N-N_0$ largest such values, and put the corresponding columns together into a set, T. We have

$$R(Q) \leq \sum_{i\in Q} R(i) \leq \sum_{i\in T} R(i). \tag{29}$$

Thus this last sum is an upper bound on the maximum reduction possible. This yields the lower bound on the objective function:

$$f(S_0) \geq \sum_{k=1}^{M} v_k - \sum_{i\in T} R(i). \tag{30}$$

This is the second lower bound. It can only be applied when there are no infinite values in v (when the reductions become infinite some of the inequalities above break down). Extending this bound to the case where there are infinities in v would be of great use in this problem and is suggested for future work.

4.2.4. Presorting of Columns. Because of the way we set up the branch-and-bound algorithm mentioned herein, a column towards the right of the cost matrix will never appear as the parent of a column is towards the left. This means that if we wish to prune a large number of nodes, we should ensure that the most important columns are towards the left of the matrix and the less important columns are towards the right. We presorted the columns of the cost matrix as follows.

(1) Rank the entries in each row of the matrix from lowest cost to highest cost. Place 1 for lowest cost, 2 for second lowest, and so forth.

(2) Find the column with the most first places (break ties with 2 s and then 3 s, etc.) and move it to the front of the matrix.

(3) Repeat steps (1) and (2) with the rest of the columns in the matrix.

This guarantees that, of any the two columns, the one to the left has won one of these minicontests against the other and in some sense is of lower cost and hence is more important. The result of this can be seen in Figure 4.

4.3. NOMAD. Nonlinear optimization by mesh adaptive direct search (NOMAD) is a software optimization package designed for numerous optimization problems. NOMAD provides a C++ implementation of the Mesh adaptive direct search (MADS) algorithm found in [11–15]. More specifically, NOMAD solves optimization problems of the form

$$\min_{x\in\Omega} f(x), \tag{31}$$

where $\Omega = \{x \in X : c_j \leq 0, j \in J\} \subset \mathbb{R}^n, f, c_j : X \to \mathbb{R} \cup \{\infty\}$ for all $j \in J = \{1, 2, \ldots, m\}$ and X is a subset of $\mathbb{R}^n$ [14]. As already mentioned, NOMAD makes use of the MADS algorithm. Nomad is essentially an iterative method that utilises blackbox functions which evaluates trial points on a mesh [16].

Each MADS iteration pertains to three steps, the poll, the search, and finally the update. The search allows for trial points to be created anywhere on the mesh, while the poll step is strictly more defined due to the reliance of the convergence on it. Ideally, the algorithm converges globally to a point $\hat{x}$ which satisfies the local optimality conditions based upon the functions defining the problem.

A loose description of this technique can be thought of as an arbitrary point selected and a local minimum found. Iteratively repeated this process leads to numerous local minimums. Providing enough iterations are conducted, a fairly accurate global minimum can be selected from this set. Now considering the problem at hand, a glass manufacturer

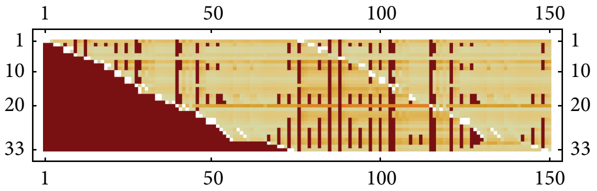

Figure 4: The cost matrix for our data. Dark is high and low is light.

may be dissatisfied with the time taken to find an exact solution using the DFS and branch-and-bound algorithms. NOMAD, however, offers the alternative of speed, albeit at the potential of an inexact solution. The results highlight such a comparison.

5. Results and Analysis

All algorithms and methods were implemented on a i7-3930k:12 cores @3.9 Ghz, 64 gigabytes of 1600 Mhz ram, 2× GTX 690 PC utilising MATLAB 2013a. We implemented the algorithms on an industrial dataset with 33 orders that produced 151 possible blank sizes from which to choose.

5.1. Complexity Analysis

5.1.1. Blank Choices and the Solution Space. As was mentioned previously, the number of blanks that can be used to satisfy the orders is established a priori. It would be appropriate to investigate how this decision might affect the size of the solution space and hence the performance of the algorithms.

Being a combinatorial optimization problem, each solution is made up of a number of different choices. The fact that more blanks are available to choose from, as well as the number that we allow to be selected, invariably increases the problem's size as there is an increasing number of possible combinations.

Using our blank generation procedure and a hypothetical data set of orders, imagine that a total of 151 blanks are produced. Referring to Table 3, by altering the number of selections from these 151 blanks we can see how the solution space grows. Immediately, one will notice that the solution space grows exponentially, to the extent so that graphing this increase without employing a log scale would not allow for effective viewing. Figure 5 is a log-scaled bar graph that allows us to visualise how the solution space grows when increasing the number of blanks that can be chosen.

5.1.2. A Theoretical Upper Bound on the Number of Blanks Generated Given the Order Lists. As we can see from the above analysis the number of blanks that can be chosen will dramatically increase the number of possible solutions. Obviously, a higher number of possible blanks to start with will also increase the size of the search space. Recall that the entity that will determine how many blanks are generated is the order list since it is from these orders that possible blanks are derived. We now attempt to find a theoretical upper bound in an effort to get some idea of the order lists' influence on the problem size.

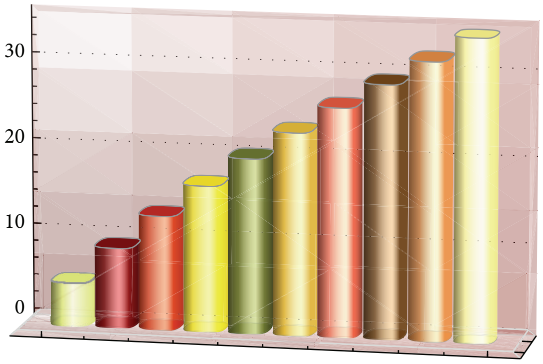

Figure 5: A log scale of solution size against choices of blanks.

Table 3: Problem size as a function of blanks.

Blanks to choose N	Number of choices
2	$\binom{151}{2} = 11325$
3	$\binom{151}{3} = 562475$
4	$\binom{151}{4} \approx 2.08 \times 10^{7}$
5	$\binom{151}{5} \approx 6.12 \times 10^{8}$
6	$\binom{151}{6} \approx 1.49 \times 10^{10}$
7	$\binom{151}{7} \approx 3.08 \times 10^{11}$
8	$\binom{151}{8} \approx 5.55 \times 10^{12}$

First, let us assume that there are M orders. An order i, for the purposes of this analysis, is represented by (x_i, y_i), where x_i is the width and y_i is the length of order i. Each blank is constructed with multiples of some order i. Therefore, the blank's dimensions of width, X, and length, Y, can be written as follows:

$$(X, Y) = (c_i x_i, k_i y_i), \quad c_i, k_i \in \mathbb{N}^{+}, \ i \in \{1, 2, \ldots, M\}. \quad (32)$$

There are a minimum width, $X_{\min}$, and a minimum length, $Y_{\min}$, that a blank can have, determined by the smallest order that needs to be cut. Similarly, a blank also has a maximum width and length ($X_{\max}$ and $Y_{\max}$, resp.,) which will be determined by equipment and other factors. Taking this into account we can say that

$$(X_{\min}, Y_{\min}) \leq (c_i x_i, k_i y_i) \leq (X_{\max}, Y_{\max}). \quad (33)$$

Only considering the width component of (33)

$$\begin{gathered} X_{\min} \le c_i x_i \le X_{\max} \\ \frac{X_{\min}}{x_i} \le c_i \le \frac{X_{\max}}{x_i} \\ \left\lceil \frac{X_{\min}}{x_i} \right\rceil \le c_i \le \left\lfloor \frac{X_{\max}}{x_i} \right\rfloor, \end{gathered} \tag{34}$$

we are able to obtain an interval for c_i, c_i's defined as the multiples of widths of order i used to construct a blank. Recall that $c_i \in \mathbb{N}^+$ and so we must utilize ceil and floor functions to ensure that c_i is a feasible value. Similarly, we can obtain a range for k_i:

$$\left\lceil \frac{Y_{\min}}{y_i} \right\rceil \le k_i \le \left\lfloor \frac{Y_{\max}}{y_i} \right\rfloor. \tag{35}$$

In order to obtain an upper bound for the number of blanks generated for an order list we now need to sum, for all orders, the product of the number of elements in the intervals for c_i and k_i:

$$\begin{aligned} \text{Blanks} = \sum_{i=1}^{M} & \left(\left\lfloor \frac{X_{\max}}{x_i} \right\rfloor - \left\lceil \frac{X_{\min}}{x_i} \right\rceil + 1 \right) \\ & \times \left(\left\lfloor \frac{Y_{\max}}{y_i} \right\rfloor - \left\lceil \frac{Y_{\min}}{y_i} \right\rceil + 1 \right). \end{aligned} \tag{36}$$

However, we can simplify (36) by noting that $\lceil X_{\min}/x_i \rceil = 1$ and $\lceil Y_{\min}/y_i \rceil = 1$ because $X_{\min}/x_i \le 1$ and $Y_{\min}/y_i \le 1$. We then continue and find an upper bound:

$$\text{Blanks} = \sum_{i=1}^{M} \left(\left\lfloor \frac{X_{\max}}{x_i} \right\rfloor \right) \left(\left\lfloor \frac{Y_{\max}}{y_i} \right\rfloor \right) \tag{37}$$

$$\le \sum_{i=1}^{M} \left(\left\lfloor \frac{X_{\max}}{X_{\min}} \right\rfloor \right) \left(\left\lfloor \frac{Y_{\max}}{Y_{\min}} \right\rfloor \right) \tag{38}$$

$$= M \left\lfloor \frac{X_{\max}}{X_{\min}} \right\rfloor \left\lfloor \frac{Y_{\max}}{Y_{\min}} \right\rfloor \in O(M). \tag{39}$$

The progression to (39) reveals that the number of generated blanks is bounded linearly by the number of orders that need to be cut.

5.2. Algorithm Performance

5.2.1. Execution Times. Table 4 is the execution times for the branch-and-bound, depth-first search, and NOMAD algorithms. As we would expect, the exact solution procedures, DFS and branch-and-bound, are heavily affected by the increase in problem size that comes with increasing the number of selectable blanks. Indeed, DFS and branch-and-bound method were only tested up to 5 and 6 selectable blanks, respectively. DFS took over 2 hours to complete with 5 selectable blanks and branch-and-bound was in excess of 3 hours for the 6 selectable blanks case. The results for all 3 algorithms can be seen in Figure 6.

Table 4: Algorithm execution times (seconds).

Selectable blanks	DFS	Branch-and-bound	NOMAD
2	0.1448	0.3362	0.5622
3	6.9312	3.4673	0.9184
4	272.97	64.0814	9.7329
5	8343.2	978.5097	1.7527
6	—	12813.420	2.8240
7	—	—	5.7266
8	—	—	1.9721
9	—	—	5.3426
10	—	—	8.2825

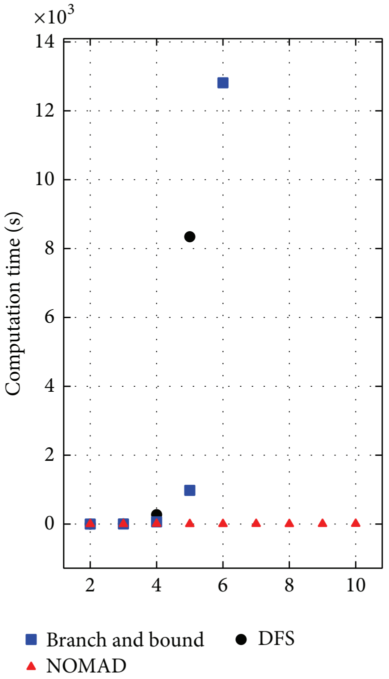

Figure 6: Scatter plot of execution times for algorithms.

We note that branch and bound, an algorithm that is an improvement on the brute force enumeration that comes with DFS, keeps a relatively low computation time up to about 4 selectable blanks. Looking at the area of interest, Figure 7, both exact solution methods were competitive with the NOMAD heuristic in terms of speed up until 3 blanks were made selectable. Thereafter, the exact solution procedures, most notably DFS, suffer as a result of the increasing problem size.

NOMAD is consistently fast in its execution, having less than 10 seconds computation time for all cases; however, a heuristic often sacrifices solution quality for speed as we will see later.

5.2.2. Branch and Bound Details. The execution time results indicate that the branch-and-bound algorithm is significantly better than the DFS algorithm for this problem; it does more work at each node. This indicates that a significant portion

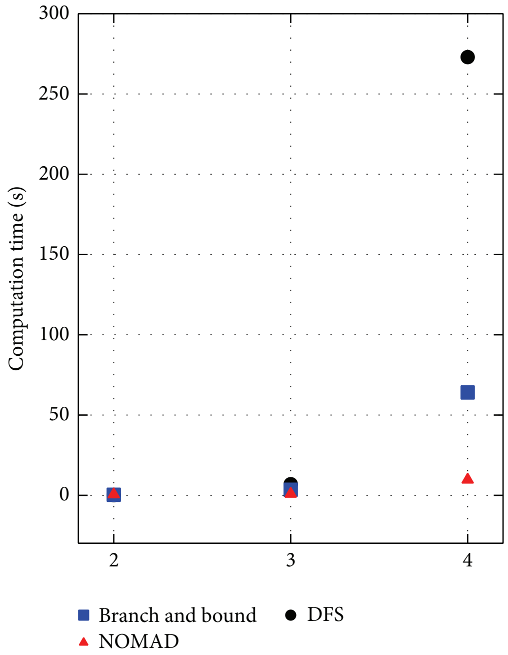

Figure 7: Scatter plot of execution times for algorithms higher resolution over smaller range (2–4 blanks).

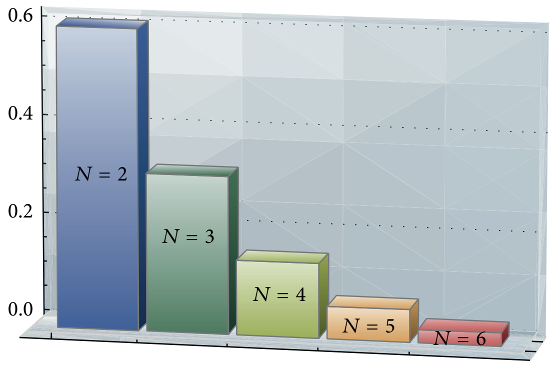

Figure 8: Fraction of search space visited by branch and bound for $N = 2, 3, 4, 5, 6$.

of the search space is being pruned. Figure 8 illustrates the fraction of the search space (leaf nodes) actually visited by the branch-and-bound method for different values of N. Observe how this fraction decreases rapidly for different values of N. It is conceivable that with better bounds it may even get close to the scale of the rapid expansion in search space size.

In the code for branch-and-bound we first implemented lower bound 1 and then for nodes that did not get pruned we tried lower bound 2. It was not at all clear that this second step would successfully prune any nodes after the first step failed. Figure 9 indicates, however, that a significant portion of nodes were pruned by the second lower bound. A third lower bound that was a slight generalization of the first one was also attempted, but did not prune many extra nodes after the first lower bound was applied, and also significantly slowed down the algorithm and so it was removed.

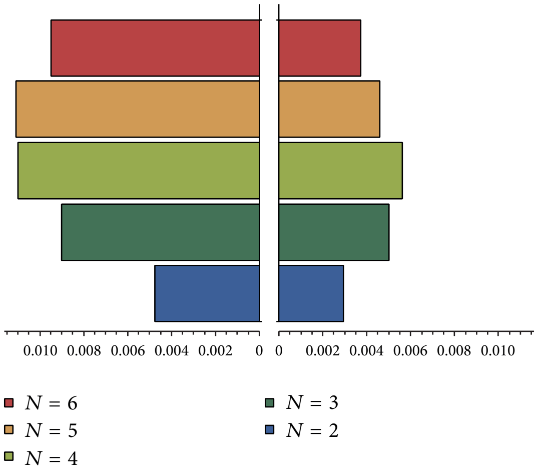

Figure 9: Fraction of nodes pruned due to lower bound 1 (left) and lower bound 2 (right).

Table 5: NOMAD solution quality.

Selectable blanks	Minimum wastage	NOMAD	NOMAD optimality error
2	114693.440	165075.128	43.927%
3	70775.004	70811.179	0.051%
4	52261.038	54371.510	4.038%
5	36800.455	60078.314	63.254%
6	29048	36451.511	25.486%
7	—	29177.763	—
8	—	19118.633	—
9	—	15719.224	—
10	—	10721.666	—

5.2.3. Solution Quality of NOMAD. Table 5 shows the wastage values obtained for DFS, branch-and-bound, and NOMAD. DFS and branch and bound are exact solution procedures and so will have the same wastage at completion, the optimal minimal wastage. These values can be found in the column of minimum wastage. NOMAD, however, is a heuristic and does not guarantee an optimal value. In fact, NOMAD performs rather erratically in solution quality, as can be seen in Figure 10. In several cases we notice significant error percentages. It would most certainly be unsuitable to obtain a solution that has an optimality of 60%. NOMAD's inconsistency makes it an unreliable tool if minimising wastage is the primary objective.

5.2.4. A Simplicity versus Optimality Consideration. A trend that we notice with the results of all algorithms is the decreasing wastage with the increasing selectable blanks. This is not surprising since if we allow more blanks to be selected we enhance our capability to cater for all the orders, reducing wastage. This can be seen in Figure 11. There is a case where NOMAD had more wastage although there had been more selectable blanks, but this can be explained by the underlying uncertainty that NOMAD has exhibited during testing.

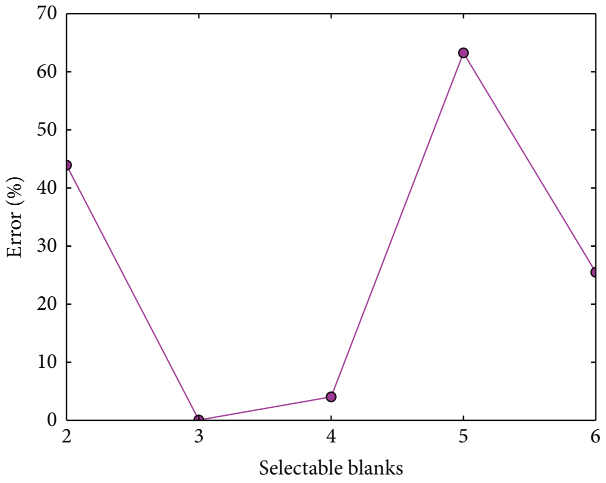

FIGURE 10: Percentage error from optimal values for NOMAD.

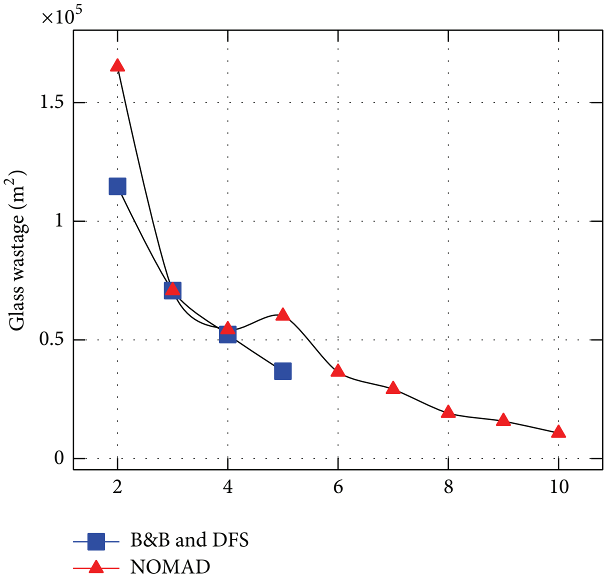

FIGURE 11: Glass wastage versus number of blanks used.

This trend of decreasing wastage is not so extraordinary. Practitioners in the glass manufacturing industry are aware of the prospect of further reducing wastage in this manner. It is rather a question of simplicity versus optimality since operating the machinery at the factory becomes more complicated as one increases the number of blanks involved in satisfying the orders.

6. Conclusions

The selection of blanks to satisfy orders holds great significance in the glass manufacturing industry. Random and heuristic methods for guessing the best blank sizes tend to result in relatively high wastage. Minimising this loss translates to a meaningful benefit for a glass manufacturing enterprise. Furthermore, it is possible that these ideas will find application in the metal, paper, and wood industries (amongst others).

Making the transition to a discrete combinatorial problem proved worthwhile. Presented with any order list we are able to generate a set of feasible blanks. It provided us with the flexibility to apply tried and tested algorithms in the field of combinatorial optimization, such as branch and bound, to optimally select these blanks. Identifying a similarity with the p-median problem effectively allowed us to formulate the problem, making it a matter of selecting columns. Some future work may involve developing a more elaborate model. For example, it would be beneficial to formulate a robust model that accommodates for different cutting machines and the option to rotate when cutting. We may also wish to include other factors into the objective function apart from wastage. In particular, it may be of interest to model the risk of an order being cancelled or changed into the stock size selection process.

It was shown that the number of blanks in the search space is bounded linearly by the number of orders, M. Nevertheless, the actual computation time is not linear in this number, because we need to choose a subset of size N from these $O(M)$ blank sizes. We expect the computational time to look like $\binom{O(M)}{N}$ and so it is polynomial in M and exponential in N.

The branch and bound implementation performed well with larger problem sizes, whilst still providing an optimal solution. The upper and lower bound estimates and presorting of columns were effective at trimming the solution space, so that for $N = 6$ only 2% of the search space had to be explored to find the optimal solution. An interesting potential improvement involves adapting the second lower bound estimate to handle cases, where there are infinities in the cost matrix, and investigating other bounding and pruning strategies. As it stands, branch-and-bound takes around 3 hours to compute the $N = 6$ case for our data and less than an hour for $N = 5$. These are feasible times for industry. NOMAD had a somewhat erratic performance when it came to solution quality, on one occasion having a 60% difference from the optimal solution. Its strength lies in fast execution times, being less than 10 seconds in all tested cases. One might be able to run NOMAD several times to try and get a sense if it is near the global minimum in the solution space and not in a grossly suboptimal local minimum. It may also be possible to fine-tune NOMAD towards solving this particular problem. Our results, however, are not entirely positive. Further work may include developing a heuristic that combines relatively fast execution times with good, consistent solution quality.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The authors wish to thank Riaan Von Wielligh from PFG Building Glass for his assistance in the problem description as well as the members of the 2013 MISG South Africa study group.

References

[1] P. C. Gilmore and R. E. Gomory, “A linear programming approach to the cutting-stock problem,” *Operations Research*, vol. 9, no. 6, pp. 849–859, 1961.

[2] P. C. Gilmore and R. E. Gomory, “Multistage cutting stock problems of two and more dimensions,” *Operations Research*, vol. 13, no. 1, pp. 94–120, 1965.

[3] P. Sweeney and E. Paternoster, “Cutting and packing problems: a categorized, application-orientated research bibliography,” *Journal of the Operational Research Society*, vol. 43, no. 7, pp. 691–706, 1992.

[4] M. Hifi, “An improvement of Viswanathan and Bagchi’s exact algorithm for constrained two-dimensional cutting stock,” *Computers & Operations Research*, vol. 24, no. 8, pp. 727–736, 1997.

[5] E. K. Burke, G. Kendall, and G. Whitwell, “A new placement heuristic for the orthogonal stock-cutting problem,” *Operations Research*, vol. 52, no. 4, pp. 655–671, 2004.

[6] M. L. Chambers and R. G. Dyson, “The cutting stock problem in the flat glass industry-selection of stock sizes,” *Operational Research Quarterly*, vol. 27, no. 4, pp. 949–957, 1976.

[7] J. Reese, “Solution methods for the p-median problem: an annotated bibliography,” *Networks*, vol. 48, no. 3, pp. 125–142, 2006.

[8] N. Mladenović, J. Brimberg, P. Hansen, and J. A. Moreno-Pérez, “The p-median problem: a survey of metaheuristic approaches,” *European Journal of Operational Research*, vol. 179, no. 3, pp. 927–939, 2007.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, McGraw-Hill, 2nd edition, 2001.

[10] A. H. Land and A. G. Doig, “An automatic method of solving discrete programming problems,” *Econometrica: Journal of the Econometric Society*, vol. 28, pp. 497–520, 1960.

[11] M. A. Abramson, C. Audet, J. E. Dennis, Jr., and S. Le Digabel, “OrthoMADS: a deterministic MADS instance with orthogonal directions,” *SIAM Journal on Optimization*, vol. 20, no. 2, pp. 948–966, 2009.

[12] C. Audet and J. E. Dennis, Jr., “Mesh adaptive direct search algorithms for constrained optimization,” *SIAM Journal on Optimization*, vol. 17, no. 1, pp. 188–217, 2006.

[13] C. Audet and J. E. Dennis, Jr., “A progressive barrier for derivative-free nonlinear programming,” *SIAM Journal on Optimization*, vol. 20, no. 1, pp. 445–472, 2009.

[14] C. Audet, S. Le Digabel, and C. Tribes, “NOMAD user guide,” Tech. Rep. G-2009-37, Les cahiers du GERAD, 2009, http://www.gerad.ca/NOMAD/Downloads/user_guide.pdf.

[15] S. Le Digabel, “Algorithm 909: NOMAD: nonlinear optimization with the MADS algorithm,” *ACM Transactions on Mathematical Software*, vol. 37, no. 4, pp. 1–15, 2011.

[16] M. A. Abramson, C. Audet, G. Couture, J. E. Dennis Jr., S. Le Digabel, and C. Tribes, “The NOMAD project,” http://www.gerad.ca/nomad.