

Experiments in Flat Glass Cutting Optimization

Joseph D. Touch

University of Scranton
Scranton, PA 18510
April 1985

Advisor: Dr. J. Beidler

Also available in ASCII text, Gzip'd Postscript, and PDF

Translated May, 1999.

ABSTRACT:

The flat glass cutting problem is in the class of NP-Complete problems, thus making an exhaustive search, of every pane in every position, impossible. Instead, an approximate solution is formed, using a combination of software techniques, involving preprocessing of the lists to be cut as well as various feedback processes within the cutting algorithms. Due to practical considerations, the methods used are further restricted to operation on a small personal computer, using an hour or less of processing time, so previously posited solutions are not applicable. This experiment will be limited to polynomial - timed solutions, and operates on large (up to 500 panes) data sets. Current results yield a daily average efficiency of up to 89%, excellent when compared to the industry yearly average of 85%.

Experiments in Flat Glass Cutting Optimization is an investigation of algorithms for the positioning of patterns in the cutting of window panes from stock glass sizes (Fig. A). Through the testing and analysis of various cutting algorithms, insight into the general problem was obtained, and possible solutions posited.

SAMPLE PATTERN FOR CUTS OF 'A', 'B', & 'C':

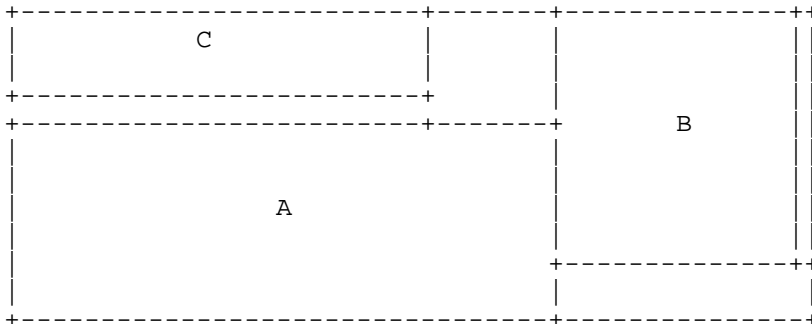


Figure A

These experiments began as a local glass cutting company, Mesko Glass, desired to automate their processes. They wanted to automate the positioning of the cuts, in order to minimize the waste generated in the cutting process. Most of the programs commonly available at the industrial level for such patterning requires expensive mainframe computers; Mesko, being a small firm, wanted to know if similar results could be obtained using their small IBM personal computer.

In general, the glass-cutting problem has been researched before, however much of this research proved inapplicable to Mesko's case. The classic, polynomial-based partial solutions rely on the cutting of enormously large stock sizes, where it would not be unusual to cut 50 to 500 panes from a single piece of stock glass[1,2]. Mesko, however, performs its cutting manually, and thus their pieces of cutting stock must be small enough to be easily handled by one or two glass cutters. Rarely are their stock pieces cut into more than 7 or 8 orders. Also, these 'classic' solutions rely on large processing times, expecting to cut the same set of orders over some period of time, thereby justifying the processing overhead[5,8,13]. Mesko must cut each day's orders as they are received, so their order list is substantially smaller, and varies greatly. The processing time of their patterning program cannot exceed one hour or so, including data entry, program execution for efficiency optimization, and print-outs adequate to serve as cutting guides.

In addition, Mesko has little funds for computer services, thus the program must be designed to operate on a very small computer, not a large mainframe. This severely limits the method of experimentation which is plausible. For instance, one area of interest, a few years ago, was linear programming. It requires vast amounts of memory space and processing time, and must be abandoned in this approach[9].

In order to determine the feasibility of this project, a preliminary program was written, in PASCAL on an IBM PC in January, 1984. This program experimented with two viable cutting methods, and helped demonstrate the feasibility of glass fitting on a personal computer. In addition, the amount of waste generated by this crude attempt closely approached Mesko Glass' overall efficiency, proving that a competitive solution was not far away. This project arose from the desire to increase that initial program's efficiency, through experimenting with the fitting algorithms.

Our desire for the solution to the fitting problem is not limited to this specific instance. The flat glass fitting problem is part of a larger set of computer science problems, the NP-Completes. These problems, by their nature, cannot be solved by 'brute force'. We could not, for instance, try every piece of glass in every position, to find the answer. If we did, our program might require literally years of computer time to solve, even for a relatively small list of panes. Instead, we are required to try other solutions. These may not lead to the ultimate answer, of absolute minimum waste, but they can come close enough to the answer, without requiring days of computer time, to provide a viable alternative. These are the solutions we will be attempting to discover, through our experimentation.

The algorithms utilized in the search for waste minimization fall into three major classifications. First are those methods which always tend to increase efficiency, due to the nature of glass cutting itself. Second are those heuristics which rely on information derived from the set of input data, where the operation of the algorithm adjusts to accommodate the fluctuations of the data. Finally are those algorithms which rely on feedback from the cutting process itself, to determine how the next stock will be cut.

In addition to these algorithms which constitute the actual cutting process, several techniques of pre-processing are used. One area of pre-processing, which has become an integral part of our research, involves the partial ordering of the panes, in order to determine a priority in selection. Since the panes are two-dimensional, simple linear sorts do not provide the proper ordering, thus other methods are being investigated.

In addition to information supplied by various coded 'test' procedures, such as a scatter-plot, interviews were conducted of Mesko's more experienced glass-cutters, to learn how they approach the problem during their work.

The experiment was performed under a list of presumptions, derived largely from interviews with professional glass-cutters. These rules include:

1. Always try to fit the largest piece in the list to be cut to the remaining plates of glass. If it doesn't fit, proceed to the next largest size.
2. Start with the smallest piece of glass available, and use larger stock sizes only when necessary. Although this will reduce efficiency, it is cost effective, since smaller stock sizes are much easier to handle, and thus less likely to shatter in transport.
3. Always place cuts orthogonally, to the outside of the plate (known as 'normalization'). While this may lead to inefficient patterns[5], it will be assumed for program simplicity.

The exploratory algorithm, designed to test the feasibility of such a cutting algorithm on a personal computer, operates on three assumptions. First, the largest order must be cut at some time; if it is cut first, there is more of a chance of finding other pieces which will use the scrap generated from that initial cut. Second, the scrap generated should be filled by the largest piece immediately, rather than fill it with two smaller pieces, which may or may not exist. Third, when cutting a piece of glass from the stock, cut it in such a way as to leave the two most usable scrap pieces (denoted as cut_type A - see Appendix A). This method uses the following algorithm:

PRELIMINARY CUTTING ALGORITHM

- I. SORT the stock list into a list of small to large.
- II. SORT the orders list into a list of large to small. (note that the sorting was performed by the naive technique of sorting by larger side first, then by smaller side within the larger.)
- III. LOOP:
 - A. get the largest order from the top of the order list.
 - B. get the smallest stock which can be used to cut that order from.
 - C. send the stock thus chosen to the recursive cutting procedure:
 1. cut the largest order (first one found by a linear search of the list) which fits inside that piece of stock, using the method denoted as 'A'.
 2. remove the order thus cut from the list of orders to be cut.
 3. set up the cut tree to reflect the cut thus made.
 4. recursively call to 1, using the remainders of stock generated by the cut as new stock sizes.

If the stock cannot be cut (the list is searched, and no fit is found) return out of the recursion.

For these tests, sample stock sizes and orders were obtained, from one month's business of Mesko. Using these orders and stock sets resulted in 79.8% efficiency. In initial investigation of the output generated by this method, an overuse of the smaller stock sizes was noticed. The efficiency seemed to be lower for these stock sizes. The initial reaction was to remove the smallest stock size from the stock list, and to re-run the program, to see if there would be any noticeable difference.

Using preliminary orders set, and the reduced stock set, resulted in 83.2% efficiency (the stock list was revised to exclude the smallest stock size, stock #1). The observations of this trial was a continued overuse of the smaller stock sizes, even though the efficiency did increase slightly. The conclusion was to remove both stock sizes 1 and 2, the smallest two sizes of the original data set.

Using preliminary orders set, and newly revised stock set, resulted in 80.7% efficiency (the stock list was revised to exclude the smallest two stock sizes, #1 and #2). The observations of this trial was a continued overuse of the smaller stock sizes, and a note that the efficiency now dropped slightly. The conclusion was that removing the smaller stock sizes was not a useful option. There must be a reason the industry uses the smaller stock sizes, other than mere handling requirements. Small stock sizes clearly must be used, but discriminately. The question is how to decide when to use them.

Reevaluating data obtained from the three above experiments, a rough graph was hand sketched of the efficiency of each piece being cut (Fig. B). A visual evaluation of the graph thus sketched showed that some of the stock sizes have 'erroneous' efficiencies.

The efficiency drops, in a generally decreasing fashion, for each stock being cut. Most of the stock being cut does not err from this steady, slightly decreasing rate.

In the first case, using the original test data, 6 of the clearly erring pieces of stock have efficiencies less than 75%. Of these, all use stock size #1. This explains the increased efficiency seen in trial 2, where stock #1 was removed from the list.

In the second case, only 3 of the stock pieces have efficiencies less than 75%, and of these both are of stock #2. This explains why removing stock #2 as well as #1 does not increase the efficiency. Not that many of the cuts executed were affected by removing stock #2.

A major concern at this point was to obtain a higher efficiency using the smaller stock sizes, to justify their use by the industry.

A new strategy was created: to recut the orders from a larger stock size, if the efficiency was 'out of line'. Temporarily, 'out of line' will be defined as having an efficiency less than 80%. This method was denoted the 'backup' method, since, if the efficiency was errant, the cut would be 'undone', or backed up, and then recut using a larger stock piece.

Preliminary analysis proved the viability of the 'backup' method. However, more than one data set must be analyzed in order to determine whether the method is viable or not.

This analysis shows that a backup on efficiencies less than 80% yields an overall efficiency of 85.3%, which is excellent, compared to reported industry standards of 85% (yearly) efficiency[13].

When in the process of backing up, if selection of the stock goes 'off the end of the list', it is best to choose the first stock in the list. This rule is based on the statistical evidence that the first stock chosen is usually the best in the list. Even if that piece is not the most efficient, it is the smallest. Since this set of cuts will generate an unacceptable level of waste anyway, it is wisest to misuse the smallest, to reduce the effect on the overall efficiency (Table C).

In order to properly test the program and how it functions, a variety of data must be utilized. In the initiation of this project, Mesko Glass Company was contacted, as a possible user of

Figure B

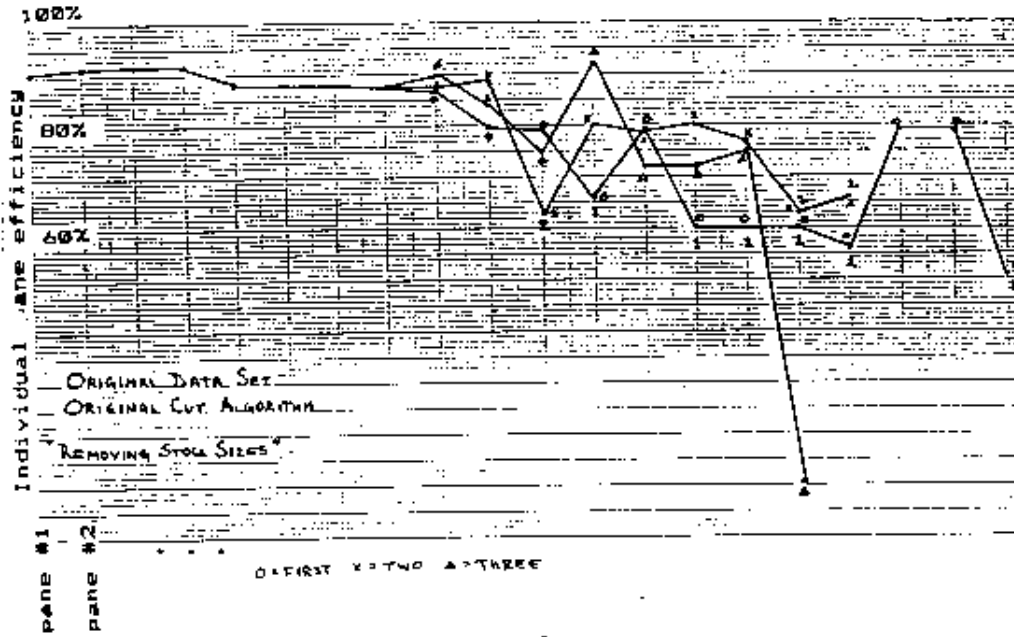


Table C

| Number of panes | Effic. % | # "off end" | New Effic. | # first is best |
|-----------------|----------|-------------|------------|-----------------|
| 60 | 71.9 | 7 | 79.9 | 6 |
| 100 | 82.8 | 8 | 84.9 | 7 |
| 150 | 88.5 | 3 | 89.0 | 0 |
| 200 | 87.4 | 1 | 87.5 | 1 |

the final product. In exchange for the final program, Mesko was to aid us in the development of the user interface of the program, and to provide us with test data, to be used in testing the functioning of the program. However, due to administrative difficulties, data from Mesko was not available in the quantities necessary to properly test the algorithms.

This development has hindered our initial experimental design plan. We had hoped to use actual industry data to evaluate our code. However, since our principle source of data was eliminated, it became necessary to obtain data from an alternate source.

In order to generate the voluminous amounts of data needed to properly test the program, it was decided to generate the data files using a random number generator. However, in order to do such, a generator had to be located and tested as being truly random.

The focus of the project was diverted to the evaluation of the available random number generators, in order to determine which would be best for our needs. Three generators were at our disposal: the MODULA MathLib0 generator (included in the runtime support package), a generator translated from a TI-58C handheld calculator, and one written by Dr. J. Beidler.

Three tests are traditionally performed on random number generators, to determine the randomness of the numbers they generated. The three tests are the Frequency test, the Serial test, and the Gap test. The Frequency test counts the occurrences of each number generated, where an even distribution shows frequency independence. The Serial test plots two successive random numbers, in (x,y) fashion. It is used to find sequential predictability in the generator. The Gap test measures the gap between a number's occurrence, and its last previous occurrence. It is designed to test the cyclic tendencies of the generator.

The TI-58C calculator generator failed all three tests; it showed definite cycles, did not generate an even distribution, and was accurately predictable. The MODULA MathLib0 generator passed all three tests successfully. It showed true randomness, and was thus suited to our needs. Dr. Beidler's generator was not tested, since a standard MODULA facility could satisfy our requirements.

It is useful to note that we may not desire a truly random or evenly random generation of panes to be cut. We may want skewed or 'normal' distributions for our tests. However, if we have a true random generator at our disposal, we can create normal or skewed distributions of any weighting.

Some questions to be addressed, when randomly generating the data files are:

- A. Should the cut list be evenly distributed, or skewed? If skewed, how so?
- B. Should the sizes generated be thoroughly random, are merely random selection of standard sizes? What are the 'standard' sizes?

C. Should the stock be generated:

1. simultaneously with the orders.
2. independently from the orders.
3. kept constant.

If the stock is generated, how should it be distributed and selected? (see A,B).

Within the program, there are provisions for plotting a cluster graph, of height/width ratio vs. area (see Appendix C). It may be beneficial for the program to recognize clustering in certain areas of the graph, such as pieces with large height-to-width ratios and large areas, so that these pieces may receive special consideration when deciding which cutting method (A,B,C,D - see Appendix A), or which order of selecting the cuts should be used. The program should be equipped to handle all cases of clustering, if and when it occurs.

A parallel issue to cluster programming is the generation of data to properly test the clustering program. These considerations further complicate the generation of new data sets.

A program has been developed to generate evenly distributed order data sets. For the initial testing, the program will generate panes whose dimensions lie between 12 inches and a maximum dimension specified by the user.

The preliminary stock data will be used as the stock data set for these initial testing purposes. A full listing of this stock set appears in Appendix B.

The number of panes generated in the cut orders file will be specified by the user. Also specified by the user is the seed integer to be used in the generation of the random numbers, facilitating user control of the generation sequence. Since the seed can be recorded, data sets can be regenerated; they need not be saved in whole.

The initial data sets used to test the performance of the program have dimensions between 12 inches and 130 inches for the longer side, and between 12 inches and 90 inches for the shorter side. Thus the pieces generated will lie between one foot square and the dimensions of the largest stock size.

In partial response to the correction of a coding bug, a warning message was incorporated to alert the operator if, in the course of finding a stock whose efficiency was over 80%, the program ran out of stock sizes to 'back up' to.

It should be noted that, in the decision making process, certain decisions cannot be correctly assumed. For example, in the case of deciding the method of sorting, how can one decide whether a piece is larger or smaller? Since there are two dimensions, are they to be sorted by larger side first?

While this may seem to be a logical method of sorting, consider that we sort the panes for a purpose: to locate those panes which are largest. We wish to cut the largest, first. However, we cannot make the assumption that the longest side implies the largest. What about largest area? What do we really

mean or intend by 'largest'?

Similarly, we backup and recut those pieces whose efficiencies are 'out of line'. What do we mean by 'out of line'? What percent of error in this judgement can we ignore as trivial?

For the final series of experiments, a total of four alternate methods of sorting the initial list of orders were devised. The original method involved sorting by longest side first, and, in the event of a tie, to sort by the smaller side. This method, called the 'long sort,' ignores any consideration of area as a criterion for sorting.

Since the efficiency is calculated by area, it seems evident that there may be merit in cutting those pieces which affect the efficiency the most, first. The method of accomplishing this is to sort solely by area, the 'area' sort.

The third sorting method, the 'four-sort', combines the consideration of cutting the largest areas first, with the desire to cut the oblong, or more 'odd' shapes first. Consider a graph, of shorter/longer dimension vs. area, relative to the area of the largest piece (Fig. D).

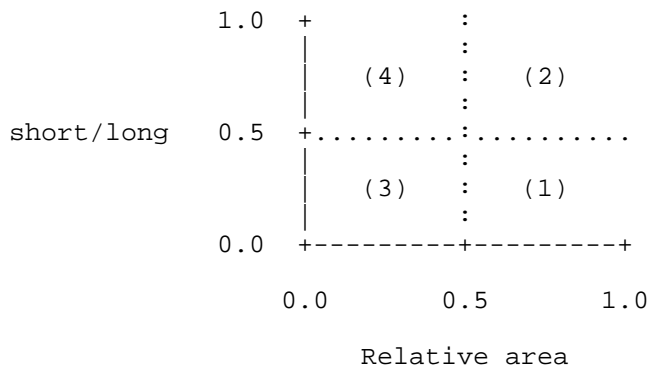


Figure D

Those pieces with large areas will be cut first (1),(2), but within that group, group (1), with short side/long side ratios less than 0.5, should be cut first. They are at least twice as long as wide, and as such are oblong. Next cut the large, 'square' pieces, (2). Then the small, oblong (3), and finally the square, small pieces (4). Within each group, an arbitrary 'long' sort is performed, to provide some ordering, although this is probably not necessary, due to the nature of the grouping. The four groups are then connected together, end to end, resulting in a ordering of (1)-(2)-(3)-(4).

Finally, a more continuous form of the discrete 'four-sort', the 'coefficient sort', was developed, by the formula:

$$[(\text{short_side} / \text{long_side}) + (1 - \text{area} / \text{max_area})] / 2$$

Where max_area is the largest area of a pane, and the result is

an average of a piece's 'largeness ratio' and its 'oblong-ness.' Since the pieces are to be sorted from low coefficient to high, the area ratio had to be inverted as (1 - area_ratio).

Given the initial algorithm, two places of sorting were possible. The 'first sort' determines the order the panes are in when the decision is made to 'cut the largest piece', which becomes the top of that list. The second ordering is in the 'filling' algorithm, where the stock and its subsequent scrap is utilized. This is when the decision is made to search the list 'to find the largest order which fits the remaining scrap.'

For the final testing, the decision to 'backup' on inefficient patterning was also optional, resulting in the following experimental design:

| first | fill | backup |
|--------|------|---------|
| | | |
| long \ | / | long \ |
| coef \ | x / | coef \ |
| four / | \ | four / |
| area / | \ | area / |
| | | x / yes |
| | | \ no |

These methods were tested in the original algorithm, which can essentially be described as a 'first fit decreasing' scheme. Each test run was performed on a fixed set of randomly generated orders, with lists ranging from 50 orders to 200 orders, in increments of 10. Each of these sizes was run with 10 data sets, the results being averaged. Thus a full test at this point involved:

4 first sorts * 4 fill sorts * 2 (backup/no backup) = 32 program types

16 data set sizes * 10 data sets per size = 160 complete data sets (50..200)

32 program types * 160 complete data sets = 5120 full program runs

5120 program runs * 30 seconds estimated average CPU time per run = (VAX 11/780, running MODULA2)

42 hours, 40 minutes of CPU time

Repeated program runs were not feasible, since our school has a single VAX to serve over 400 Computer Science majors, various other users, such as accounting classes and word processing courses, and faculty. In fact, much of the time spent involved circumnavigating the system constraints designed to prevent any program from using as much CPU time and space (25 Megabytes-to store raw data and output) as was necessary for only one of these test runs.

The data from these experiments were tabulated and several averages reported, one average for each data set size. The two most significant figures were average CPU time, and average

patterning efficiency. In addition, two other values which measure the algorithm's effectiveness were the average number of 'backups' performed while cutting the panes, and the number of times the backing up resulted in 'going off the end' of the list of stock sizes. This data appears in Appendix D.

In order to effectively evaluate the voluminous data from these trials, an algorithm was designed which correlated various values, using the method of least squares. In order to determine which type of function best approximates the relationship of the two correlated variables, not only were the actual values correlated, but their logarithms, inverses, etc. The following were determined to provide an adequate sampling of the probable relationships the variables could contain:

| Correlate | | obtaining slope (m) and intercept (b), which then represent: | Giving a relationship of: |
|-----------|----------|--|---------------------------|
| X | vs Y | | |
| X | Y | $Y = m * X + b$ | linear |
| X | log(Y) | $Y = e^{mX} * e^b$ | exponential |
| log(X) | Y | $Y = \log(b * X^m)$ | logarithmic |
| log(X) | log(Y) | $Y = b * X^m$ | polynomial |
| log(X) | 1/Y | $Y = 1 / \log(b * X^m)$ | inverse logarithmic |
| 1/X | log(Y) | $Y = e^{(m / X)} * e^b$ | inverse exponential |
| X*log(X) | Y | $Y = m * X * \log(X) + b$ | nlog(n) |
| X | Y*log(Y) | $Y * \log(Y) = m * X + b$ | (added for symmetry) |

Each set of data pairs to be evaluated were correlated by these eight methods. The method with the highest correlation coefficient (closest to 1 or -1) best expresses the relationship between the coordinates. Note, however, that this determines only the major relationship between the pairs of data. The data may be related in multiple ways, such as both linear and exponential, i.e.:

$$Y = 3 * X + 0.1 * X^{4.1}$$

In this case, the linear portion of the curve is dominant for small values of X, while at larger values the exponential nature of the expression is evident. For these purposes, finding the dominant relationship is sufficient.

It was predicted, and experimentally proven, that the best correlations would be data set size (number of panes) vs. efficiency, and data set size vs. CPU time. The efficiency follows an inverse exponential with respect to set size, and the CPU time follows a polynomial relationship to set size.

Efficiency is an inverse exponential for obvious reasons.

As the data set size increases, the chances of finding orders to be cut which will utilize the scrap generated by other cuts increases, until a maximum is reached. The maximum here is expressed by the constant e^m .

The CPU time is a polynomial by program design. The desire was to find a polynomial based algorithm for relatively efficient patterning, by testing various polynomial-timed algorithms. The highest power of the polynomial is 'm'.

A condensed listing of these factors (of polynomial time constant, and efficiency asymptote), appears in Table E. Program method is listed as first sorting method, fill sorting method, where L = long sort, C = coefficient sort, F = four sort, and A = area sort. A full listing of the correlations appears in Appendix E.

Table E

| Program Method | With backup | | Without backup | | DELTAS | |
|----------------|-------------|--------------|----------------|--------------|----------|--------------|
| | CPU poly | effic. limit | CPU poly | effic. limit | CPU poly | effic. limit |
| L L | 1 2.56 | * 90.6 | 17 2.77 | 89.1 | .21 | -1.5 |
| C L | 2 1.78 | 77.6 | 18 2.08 | 77.9 | .30 | -0.3 |
| F L | 3 2.30 | 85.6 | 19 2.51 | 83.9 | .21 | -1.7 |
| A L | 4 1.93 | * 90.3 | 20 2.15 | 88.8 | .85 | -2.5 |
| L C | 5 1.73 | 79.2 | 21 2.17 | 76.0 | .44 | -3.2 |
| C C | 6 2.20 | 84.4 | 22 2.69 | 76.4 | .49 | -8.0 |
| F C | 7 2.00 | 83.1 | 23 2.42 | 77.0 | .42 | -6.1 |
| A C | 8 1.69 | 84.3 | 24 2.13 | 75.4 | .44 | -8.9 |
| L F | 9 2.31 | 85.1 | 25 2.53 | 77.9 | .22 | -7.2 |
| C F | 10 2.12 | 74.7 | 26 2.35 | 69.8 | .23 | -4.9 |
| F F | 11 2.23 | 83.5 | 27 2.64 | 77.7 | .41 | -5.8 |
| A F | 12 2.16 | 84.8 | 28 2.42 | 79.6 | .26 | -5.2 |
| L A | 13 1.95 | * 90.4 | 29 2.14 | 90.3 | .19 | -0.1 |
| C A | 14 1.88 | 78.7 | 30 2.12 | 78.7 | .24 | 0.0 |
| F A | 15 2.14 | 89.1 | 31 2.28 | 86.6 | .14 | -2.5 |
| A A | 16 2.60 | * 91.0 | 32 2.80 | 88.4 | .20 | -2.6 |

This table gives some very interesting results. First, all of the efficiency limits dropped when the backing up process was removed, giving more than a single instance of improvement. Backing up seems, in every case, to be at least somewhat beneficial. Second, the four highest efficiencies (*) are for the area and long sorts, in various combinations, thus proving the superiority of these sorts, for this algorithm only. Other algorithms may give different results, so this would not be a reason for abandoning the other sorting techniques.

In the area of timing analysis, several startling results were obtained. First, all of the timings proved polynomial, as expected, but of orders between 1 and 3, whereas the predicted limit would be 2 (a squared polynomial relationship). The lower bound of 1 is expected. In addition, the backup algorithm was

consistently 'squarer', while the non-backup algorithm behaved more as cubed. This implies that the true timing has both squared and cubed terms, and the backing up process adds another squared term, thus increasing the squared term's influence over the behavior of the polynomial.

This is consistent with the timing analysis (Appendix A). The algorithm itself has cubed timing, while the sorts all have squared timing (worst case). Since the backup method executes more sorts, every time it 'backs up,' it should be more influenced by the squared term of sorting, and thus show a 'squarer' curve. The non-backing up algorithm did not execute as many sorts, and would thus have a smaller squared term, permitting the cubed term to dominate the curve.

Note that while the polynomial was of a higher degree without backing up, backing up did increase the CPU time required to run the algorithm, in every case, as would be expected.

The first variations of this experimental design involved altering the set of stock sizes, from which all panes are cut. One variation, stock set 'Two', involved halving the largest piece of the original stock set, until it was too small to be usable. Another variation of the stock data set was a totally randomly generated set. Since the previous experimental design required over 40 hours of CPU time to run, the design was restricted to the starred (*) sorting methods which proved most viable above. The condensed results of these two tests appear in Tables F and G, while the fully correlated results appear in Appendix E. A listing of the stock data sets, in full, appears in Appendix B.

Table F: Second, Created stock set

| Program Method | With backup | | Without backup | | DELTAS | |
|-------------------|-------------|-----------------|----------------|-----------------|-------------|-----------------|
| | CPU poly | effic. limit | CPU poly | effic. limit | CPU poly | effic. limit |
| L L | 1 2.47 | 88.0 | 17 2.67 | 86.2 | .20 | -1.8 |
| A L | 4 1.98 | 87.6 | 20 2.12 | 87.5 | .14 | -0.1 |
| L A | 13 2.03 | 87.9 | 29 2.14 | 88.1 | .11 | +0.2 |
| A A | 16 2.59 | 89.0 | 32 2.71 | 87.5 | .12 | -1.5 |

Table G: Third, Random stock set

| Program Method | With backup | | Without backup | | DELTAS | |
|-------------------|-------------|-----------------|----------------|-----------------|-------------|-----------------|
| | CPU poly | effic. limit | CPU poly | effic. limit | CPU poly | effic. limit |
| L L | 1 2.49 | 88.2 | 17 2.72 | 85.2 | .23 | -3.0 |
| A L | 4 1.99 | 87.5 | 20 2.13 | 86.4 | .14 | -1.1 |
| L A | 13 2.02 | 90.0 | 29 2.17 | 87.3 | .15 | -2.7 |
| A A | 16 2.59 | 88.3 | 32 2.72 | 86.0 | .13 | -2.3 |

Note how both of these trials generate similar results, with efficiencies maximized at about 90 %, or 10 % waste. It seems that, within reasonable limits, the contents to the stock set is largely inconsequential.

For all of these trials, the stock set had to first be sorted, just as did the orders set. The algorithm performed only the long sort on the stock list. While it may seem presumptive not to vary the stock sorting as well, it was determined early in the experiment, by coincidence, that all four of the sorting techniques yield identical orderings. This was true for the first stock set only, but since the number of stock pieces was small, the ordering was deemed inconsequential, provided there was a general trend of small to large.

It is also useful to note the accuracy of these figures. In addition to the averages computed, standard deviations, of N-1 weighting were also calculated. Representative values of these SD's appear in Table H.

Table H

| Data Item: | Range of SD: |
|------------|--------------|
| Efficiency | 0.8 % - 3.0% |
| CPU time | 2 % - 15 % |
| Backups | 50 % - 1000% |
| Off end | 50 % - 200 % |

From these figures, we can assume that the efficiencies are relatively stable, for a constant orders set size. The CPU time similarly maintains a fairly tight average, thus these two averages can be used with some degree of confidence, although it should be remembered that they are averages, not actual data.

The number of backups seemed to remain unpredictable; it was not related specifically to the number of orders processed, nor was it related to the efficiency. In fact, within the set of data from one sorting method, the number of backups seemed relatively constant.

The number of times the search for a more efficient stock size went 'off the end' of the list of available stock was similarly unpredictable. It fluctuated wildly, even within one sorting method; in fact, some of the SD's were as much as twice the value of the average. This number will exceed the number of backups on occasion, contrary to common sense. This is because it is common to begin the search for a stock size with the last item in the stock list. When this size fails to yield an acceptable efficiency, the backup immediately fails 'off the end' of the list. Due to the current implementation, this increments the 'off end' count, but not the 'backup' count.

The resultant graphs appear in Appendices F & G. Notice in these graphs how, visually, the efficiency seems to approach a limit (Appendix G), and how the CPU time graphs behave, some rising up more quickly than others. The graphs may be overlaid for comparison.

Some other methods suggested for experimentation, both by the results generated here, and by the currently available literature (see bibliography) include:

1. Selectivity in choosing cut method. The present algorithm utilizes only one of the four possible orientations. Other orientations may be more efficient, if the selection process were reconfigured.
2. A thorough research of the variations in the generation of the stock set, studying not only what combinations yield the best efficiencies, but also how the industry cost of the different sizes (price per square foot) would affect the selection of stock sizes. While overall waste should be reduced, a less efficient (area-wise) pattern may be more cost effective to the company.
3. A more thorough testing of the algorithms, using more data sets. Averages presented here were of ten data sets; this is not as statistically significant as would be desired.
4. Testing of the algorithms in an industrial setting, with actual data sets and stock sets as used daily by a company in operation. This kind of field testing is the only true test of an algorithm's validity.
5. Break the process into two 1-D problems. The result is commonly known as the 'strip-method'.
6. A fine tuning of the methods presented here, via searching all stock sizes, and choosing the most efficient at each decision, rather than 'backing up', for instance.

This experiment, while not as conclusive as desired initially, has shown very promising exploratory results. Several sorting methods have been investigated, some ruled inefficient, others surprisingly viable (especially the long and area sorts). The location of the sorting doesn't seem to affect the performance of the algorithm significantly; subsequent experiments may abandon this approach.

Other information has been obtained in the area of optimal stock sizes. While smaller, pre-cut stock sizes are considered generally wasteful, their use is necessary for handling ease by the glass cutters, and have proven to be critical in waste minimization.

The main accomplishment of this experiment has been to show that some very simple heuristics can yield industry-rivaling results. This is also the first time the issue of polynomial timing of the cutting algorithm has been given first consideration when designing a solution, resulting in an algorithm which can easily handle both small and large numbers of orders, in very small times. This algorithm, with all its testing procedures and

monitors built in, runs 20 times faster than similar algorithms, on similar data sets[8].

I would like to thank the University Library for locating the necessary research materials, Mr. Parker, of the Computer Center for allocating the required space and privileges to run the program, and Dr. Beidler for his guidance and direction. Finally, I would like to thank the students of the University, for their input and evaluations of this project, and for their patience during the many times these programs brought the system down.

APPENDICES

- A Timing analysis and proof of NP nature
- B stock data sets used
- C sample clustergraphs and pattern outputs
- D raw statistical data
- E data correlations
- F CPU time vs. number of panes graphs
- G efficiency vs. number of panes graphs

BIBLIOGRAPHY

1. Adamowicz, M. & Albano, A. "A Solution to the Rectangular Cutting-Stock Problem." *IEEE Trans. on SMC* 6, 4. (Apr. 1976), 302-310.
 2. Albano, A. & Orsini, R. "A Heuristic Solution of the Rectangular Cutting-stock Problem." *The Computer Journal* 23, 4. (1980), 338-43.
 3. Baker, B. S. & Schwarz, J. S. "Shelf Algorithms for Two-dimensional Packing Problems." *Society for Industrial and Applied Mathematics* 12, 3. (Aug. 1983), 508-28.
 4. Bentley, J. L. "An Experimental Study of Bin Packing." (unpub). (1984)
 5. Christofides, N. & Whitlock, C. "An Algorithm for Two-dimensional Cutting Problems." *Operations Research* 25, 1. (Jan.-Feb. 1977), 30-44.
 6. De Cani, P. A. "A Note on the Two-dimensional Rectangular Cutting-stock Problem." *Journ. of Operational Research Society* 29, 7. (1978), 703-6.
 7. Diegel, A. & Bocker, H. J. "Optimal Dimensions of Virgin Stock in Cutting Glass to Order." *Decision Sciences* 15. (1984), 261-274.
 8. Dyson, R. G. & Gregory, A. S. "The Cutting Stock Problem in the Flat Glass Industry." *Operational Research Quarterly* 25, 1. (1974), 41-53.
 9. Gilmore, P. C. & Gomory, R. E. "A Linear Programming Approach to the Cutting-stock Problem." *Operational Research Soc.* 9. (1961), 849-59.
 10. Herz, J. C. "Recursive Computational Procedure for Two-Dimensional Stock Cutting." *IBM Journal of Research and Development* 16. (Sep. 1972), 462-69.
 11. Johnson, David S. "Fast Algorithms for Bin Packing." *Journal of Computers and Systems Sciences* 8. (1974), 272-314.
 12. Lai, T. & Sahni, S. "Anomalies in Parallel Branch-and-Bound Algorithms." *Communications of the ACM* 27, 6. (Jun. 1984), 594-602.
 13. Masden, O. B. "Glass Cutting in a Small Firm." *Mathematical Programming* 17. (1979), 85-90.
-