Fuel Cell Assembly Process Flow for High Productivity

Problem Presenter

Ram Ramanan
Bloom Energy

Problem Participants

Ibrahim Diakite, University of Texas, Arlington
Tyson DiLorenzo, Rensselaer Polytechnic Institute
David A. Edwards, University of Delaware
Brooks Emerick, University of Delaware
Rui Fang, University of Delaware
Fangxu Jing, Georgia Institute of Technology
Longfei Li, University of Delaware
Jennifer Miller, University of Delaware
Mark Panaggio, Northwestern University
Angela Peace, Arizona State University
Christopher Raymond, University of Delaware
Yu Sun, University of Delaware
Esther Wolff, Rensselaer Polytechnic Institute
Matt Zumbrum, University of Delaware
and others...

Twenty-Eighth Annual Workshop on Mathematical Problems in Industry
June 11–15, 2012
University of Delaware

Section 1: Introduction

Bloom Energy manufactures power modules based on fuel cell technology. These are built up in a modular fashion from basic components that are supplied to Bloom by several vendors. The basic components that Bloom receives from its vendors are not necessarily perfect, but vary from component to component. Each basic component has a number of physical properties which would each ideally take on a fixed value, but in reality can come from a range of values. For best performance, these properties should be matched for the components within a power module, subject to a number of rules which Bloom has developed, both through basic modeling and through manufacturing experience. In addition, a few components have properties which limit their possible placement within a cell assembly.

Currently, these rules for assembling the basic components are implemented by hand by experienced personnel: given a list of components currently in stock, which includes information about the characteristics of each, a production schedule (a plan for how to assemble the current stock of components into power modules) is designed which attempts to maximize the use of the available components. The production schedule is designed more or less by hand. Bloom's production has grown rapidly, and this strategy, which worked well initially when dealing with a small number of components, is becoming less workable when dealing with a stock of tens of thousands of components at any particular time. Bloom expects steady growth, and continuing to depend on a production schedule generated by hand is projected to quickly become infeasible.

Bloom would like to maximize productivity and minimize the number of "delayed" components (currently about 10–15% when the schedule is designed by hand), while maintaining the performance and reliability of their final product. In this manuscript we use the term "delayed" to refer to components that cannot be assembled under the existing schedule. Hence these components must be held in inventory until new components arrive, which slows down production and incurs associated costs.

Bloom came to the MPI 2012 workshop with several questions related to potential modifications of the assembly rules currently in use, as well as the question of how to forecast the number of completed power modules based on the number of incoming parts. It became clear early in the week that in order to answer any of these questions, what was really needed was a way to quickly and automatically generate a production schedule. The assembly process proceeds in stages: the basic components are assembled into stacks, then the stacks are assembled into columns, and then the columns are assembled into boxes. At each stage, Bloom has a set of assembly rules for how the different units can be combined. In principle, a discrete optimization problem could be posed to maximize the production of boxes from a given stock of basic components, subject to all the rules in place at each assembly stage. However, this problem is far too large to tackle directly by computer, even for current production volumes.

Therefore, most of the participants concentrated on a simplified subproblem, that of assembling stacks into columns, subject to a subset of the rules actually used by Bloom.

The hope was that if this simplified problem could be effectively (*i.e.*, quickly and approximately, but to good accuracy) solved, then variations of the same idea could be used for the other assembly levels. This simplified problem of generating columns from stacks can in fact be posed as an integer programming problem; unfortunately, even this subproblem turned out to be far too large to attack directly. Therefore, the group split up into subgroups that tested a variety of approaches for generating columns.

Most of the subgroups worked on strategies for building up columns in an iterative fashion. The strategies were implemented on real data provided by Bloom. All of these iterative approaches ran quickly and produced delay rates which were often competitive with or superior to the benchmark, which we considered to be 15%. None of the heuristic strategies was clearly superior to all others for all sets of inputs, but since they all run very quickly, and scale reasonably well with the system size, in principle they could all be implemented, with the best result chosen for any particular set of inputs; presumably other heuristic strategies could be implemented as well, and similar strategies implemented for the other assembly levels.

An outline of the rest of the report follows. Section 2 discusses terminology and outlines the assembly rules that Bloom actually uses. Section 3 discusses some of the underlying ideas common to all the heuristic assembly strategies that were implemented. We note in particular that all of these heuristic strategies are deterministic. Section 4 compares and contrasts the different deterministic strategies for combining stacks to build columns. Sections 5 and 6 outline the results of those deterministic strategies. Section 7 presents the linear programming framework in which the problem can be posed. Section 8 presents results from a random assembly procedure, which explored the configuration space randomly, rather than deterministically. Section 9 discusses various postprocessing strategies, in particular the use of a genetic algorithm strategy to improve results. Finally, we conclude with some general observations about the results so far and the work that remains to be done.

Section 2: Preliminaries

s=8
7
6
5
4
3
2
1

Figure 2.1. Schematic of eight-stack column assembly.

In order to simplify later presentation, we begin by presenting some definitions used to characterize Bloom's technology. The piece of equipment that Bloom actually ships is called a *box*. Each box contains c *columns* where the power generation takes place. (In current designs, $c = 8$.) The columns themselves are assembled from s *stacks*. (With current technology, s, the *column height*, is 8 or 10.) Each stack in the column is assigned a position number, with 1 being at the bottom and s being at the top. The case with eight stacks is shown in Fig. 2.1.

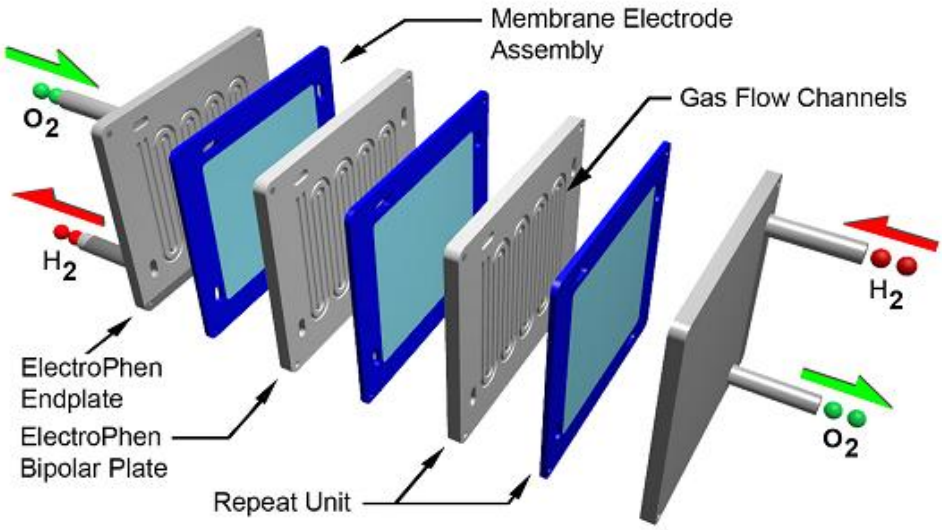


Figure 2.2. Schematic of stack assembly.

Each stack is made up of R *repeating units*, which are made up of an interconnect (IC) and a cell (see Fig. 2.2). (With current technology, R can range from 10–50.) Only the ICs are characterized in ways that affect the assembly.

Each shipment of ICs from a vendor is identified with a value of the parameter A (A1 in Bloom's parlance), which is related to the area of the pores in the IC. Different vendors ship ICs with different values of A, which can range between 0.14 and 0.24. The value of A given by the vendor for any shipment is the median value of all the ICs in that shipment. Hence it is impossible to know the value of A for any particular IC. Depending on the value of A, Bloom identifies the shipment with a *bin*. Currently, the bins are numbered from 0 to 9 using the following rule:

$$A \in (0.14 + 0.01M, 0.15 + 0.01M) \qquad \Longrightarrow \qquad \text{shipment in bin } M. \tag{2.1}$$

The vendors also provide another measurement (A2) characterizing the ICs, but we were told to ignore it for the purposes of this project.

The goal of the project is to minimize the number of *delayed stacks* in the assembly process. Here "delayed stacks" means those components (repeating units, stacks, or columns) that cannot immediately be used in a box, but must wait until a future shipment arrives in order to be used.

The tens of thousands of ICs are assembled into thousands of stacks. When assembling a stack, choosing ICs with similar values of A optimizes the flow through the fuel cell. Hence one prefers to use ICs from a single bin and a single vendor. Fortunately, in a typical month, tens of thousands of ICs arrive to be assembled. Hence it is quite easy to assemble the ICs into stacks. In particular, if we ignore the "same-vendor" requirement (which Bloom is known to relax), then the maximum number of delayed ICs is reached if exactly $R-1$ ICs were left in each of the 10 bins. But this corresponds to (at most) a few percent of the monthly volume. Hence for the purposes of this project we ignored stack assembly and treated the stack as the lowest-level building block.

Once a stack is assembled, it is assigned a value of A by taking the median value of A for its component ICs. Then the stack is assigned a bin per (2.1). Note that if the stack is assembled from ICs from a single bin per the goal, the bin for the stack would be the same as for each individual IC.

Three additional measurements are taken for each stack:

1. In the vast majority of cases (95%–98%), the stack is within normal operating parameters. However, there are stacks whose shape or electrical properties restrict its position placement in the column. The variable G (G2 in the Bloom parlance) characterizes those properties of the stack.
2. T (G1T in the Bloom parlance) is a positive number that characterizes the curvature of the top of the stack.
3. B is a positive number that characterizes the curvature of the bottom of the stack. It is denoted as G1B in the Bloom parlance and is negative. However, for algebraic simplicity we use $B = |\text{G1B}|$ instead.

Then the thousands of stacks are assembled into hundreds of columns according the the following rules:

Assembly Rules.

1. If a stack's shape property is anomalous, it must be placed in position s.

2. If a stack's electrical properties are anomalous, it must be placed in a position less than or equal to $s/2$.
3. Denote T_j as the value of T for the stack in position j, and similarly for B_j. Then

$$T_j + B_{j+1} \leq q, \qquad j = 1, 2, \ldots, s-1, \tag{2.2}$$

   where q is a tolerance value (initially taken to be 400). Equation (2.2) ensures that the sum of the curvatures is small enough so that gas leakage does not occur when the stacks are joined.
4. Again motivated by flow considerations, the following bin requirements also hold:
   (a) The ideal case is for all stacks in a column to be from the same bin; one would want at least 50% of the columns to be of this type.
   (b) Up to 40% of the columns can contain stacks from two bins, as long as the lower half contains stacks from bin j, and the upper half contain stacks from bin $j+1$.
   (c) Similarly, up to 10% of the columns can contain stacks from three adjacent bins, as long as the stacks are ordered in nondecreasing bin number.

Once a column is constructed, it is given a bin value which is the sum of the bin values of each of its component stacks. Then the hundreds of columns are assembled into dozens of boxes. The ideal case is for all columns in a box to have the same bin value. The secondary rule we were given was to have all the columns' bin values within ± 1 of each other. But this is inconsistent with rule #4 above, which would force the column bin values to be quantized over larger ranges. Instead, a better rule used is that all the columns in a box must come from one of the categories in rule #4.

In order to reduce the number of variables under consideration, we made two simplifications. First, we say that

$$\text{anomalous shape in stack } k \qquad \Longrightarrow \qquad T(k) = Q, \qquad Q > q. \tag{2.3a}$$

With the definition in (2.3a), we see that (2.2) will never be satisfied for any stack with an anomalous shape. Hence any stack with an anomalous shape will have to be placed at the top of a column, and (2.3a) makes rule #1 redundant. Similarly, if we define

$$\text{anomalous electrical property in stack } k \qquad \Longrightarrow \qquad B(k) = Q, \tag{2.3b}$$

then (2.2) will never be satisfied for any stack with an anomalous electrical property. Hence stacks with such properties will have to be placed at the bottom of a column. This is more restrictive than rule #2, but given the small number of stacks with these properties, this additional restriction should not appreciably affect the larger problem.

With several thousand stacks to consider at any one time, any exhaustive searches of possible column configurations will be computationally infeasible (for further discussion, see §7). Therefore, we largely focused on simple and fast algorithms that could yield nearly-optimal results. When doing so, we found that there were many separate areas to consider:

1. The *pool protocol*, which determines the initial set of stacks to assemble.

2. The *size protocol*, which dictates how one chooses the sizes of subunits to assemble.
3. The *assembly protocol*, which dictates how one chooses the next two subunits to assemble.
4. The *scission* (or *chopping*) *protocol*, which dictates whether assembled subunits are broken down into smaller parts.

We will discuss these facets in detail in the following sections.

Section 3: Pool and Size Protocols

Pool Protocols

There are several ways to consider choosing the initial pool of subunits to assemble. The most obvious is to use one of the bins already identified by Bloom in (2.1) as our pool. Initially we thought that to reduce the number of delayed stacks, we would have to do something more complicated. However, when results came in that showed few delayed stacks using stacks from just one bin (see §§5, 6), this issue took on less importance. For completeness, we present the ideas discussed.

One idea is to work with ranges of A, rather than bins, which are arbitrarily chosen. A single bin has width 0.01, and we want to keep that tolerance on A. Hence we could replace rule #4 with

$$\max_j A_j - \min_j A_j \leq 0.01 \tag{3.1}$$

for any column. (Note that the extrema are taken over positions.) The rule given by (3.1) is more flexible than the bin-based rule, since it allows the consideration of ranges that span more than one bin. Hence its results should be comparable to the same-bin rule, and better than the different-bin rule, given the fact that the A value for any particular IC is unknown (only the median of a shipment is known).

Suppose we want to select N stacks for our initial pool. The first question is the selection of the number N itself. Suppose that we have M stacks that satisfy (3.1). We could select $N = M$ and use the entire pool. However, with a value of A given for each stack, we can choose different ranges that do not correspond to the bins. Hence it may be better to start with a smaller number, thinking that any unused stacks from the first iteration can be combined with stacks from a neighboring range to form a new batch for column formation.

But if $N < M$, how small should we make it? The least possible is $N = cs$, which corresponds to a single box, after which we could add additional stacks as needed if we were unable to assemble a full box. Or we could take integer multiples of cs. But which one? The largest that will fit in the range? $N = M/2$? Other choices?

Then there is the selection of the range itself. We could start with the range with the largest M, thinking that any unused stacks could then be incorporated into a new range later on. Or we could start with the range with the smallest M, thinking that this somehow would be rate limiting. Or we could start with a range including one of the endpoints $A = 0.15$, $A = 0.25$, since such extremal values of A fit in few ranges.

Size Protocols

Once the assembly process begins, we need to decide what size subunits are available for assembly at each step of the algorithm. There were two ideas investigated at the workshop.

The first was the *one-size protocol*. In this approach, the assembly process was divided into phases. In each phase, subunits of identical size were paired up, then removed from the pool for that phase. At the end of each phase, any unused subunits were discarded and the remaining joined subunits formed the pool for the next phase. Hence in the simplest case, the first phase made pairs, the second quartets, and the third octets, which are full columns if $s = 8$.

The second was the *all-size protocol*. In this approach, assembly took place in one phase. Once two subunits were combined, the resulting larger subunit remained in the pool to be joined again. Hence at any point subunits of various sizes were available to be combined. The only time a subunit was removed from the pool was when it formed a column.

Section 4: Deterministic Assembly Protocols

Both T and B can vary from stack to stack. In particular, they can be modeled as independent variables taken from a normal distribution. Some characteristic values are

$$T \sim \mathcal{N}(215, 50), \qquad B \sim \mathcal{N}(180, 70). \tag{4.1}$$

Under these circumstances, the default tolerance of $q = 400$ in (2.2) is extremely tight, and in the next section we will discuss the sensitivity of the number of delayed stacks to adjustments in the tolerance.

Given this tight tolerance, we want to optimize the assembly of the columns. However, with scores of stacks available from each bin, an exhaustive search was found to be very time-consuming. Therefore, we came up with several ideas on how to assemble the subunits efficiently (though perhaps not completely optimally). In this section, we discuss only deterministic assembly algorithms; the discussion of random assembly algorithms is delayed until §8.

Single-Step Algorithm

Several groups used *single-step algorithms*. These algorithms simply attempt to find the best match for a particular subunit. They do not consider whether the resulting larger subunit would be easy or hard to join together in the next step. These algorithms have the advantage of being fast, but may not achieve the true minimum number of delayed stacks. However, if it is an improvement to the manual system currently in use, it will be useful.

The algorithm proceeds as follows. We assume that we are given N stacks initially, and let $S = \{1, 2, \ldots, N\}$. For each element $k \in S$, define a triple $\{T(k), B(k), L(k)\}$. Here $L(k)$ will denote the length of element k, and initially all the elements are single stacks. We considered two possible ways to do the assembly.

Top Step. Define the element index t^- as follows:

$$T(t^-) = \min_S T(k). \tag{4.2}$$

We then define S_B to be the set of all possible stacks that can bind with stack t^- and still satisfy the tolerance:

$$S_B = \{k \in S : B(k) + T(t^-) \le q,\ k \neq t^-\}.$$

It is most desirable for later stages to match elements with large values of B to those with small values of T. Hence we define the element index b^+ as follows:

$$B(b^+) = \max_{S_B} B(k). \tag{4.3}$$

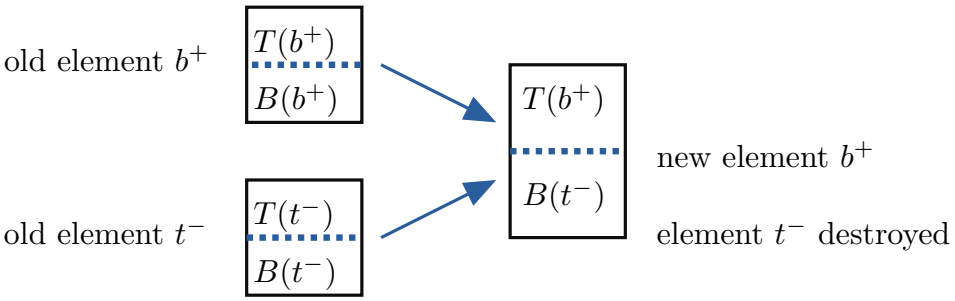


Figure 4.1. Combining of two stacks using top method.

(In other words, b^+ has the largest value of B of any subunit which can be combined with t^- and still meet the tolerance.) Then join subunit b^+ and t^- together (see Fig. 4.1).

Mathematically, we remove t^- from S and redefine the triplet corresponding to b^+ as follows:

$$\{T(b^+), B(t^-), L(t^-) + L(b^+)\}.$$

(In practice, some of the codes may have removed b^+ instead.)

The other way to do the assembly was using the **Bottom Step** method, which simply switches the roles of top and bottom from the top step. In particular, we have

$$B(b^-) = \min_S B(k), \quad S_T = \{k \in S : T(k) + B(b^-) \leq q,\ k \neq b^-\}, \quad T(t^+) = \max_{S_T} T(k).$$

In other words, one finds the stack with the largest value of T that can be combined with b^- and still meet the tolerance.

These two methods actually provide four separate single-step assembly options. One can assemble the stacks using only the top method, only the bottom method, or alternating the two, with either the top or bottom method going first. Given the fast computation times involved, several codes used all four of these methods and returned the solution that used the most stacks.

Two-Step Algorithm

Several groups worked on implementing a *two-step* method. This method tried to optimize the above algorithm to create subunits which would be easier to bind at the next level.

There were several algorithms under consideration. In one, we replace the definition of b^+ in (4.3) with

$$T(b^+) = \min_{S_B} T(k).$$

In other words, we look at all the elements which can be feasibly joined with t^-. Since smaller values of T can bind with more elements at the next step, we choose to assemble t^- with that element in S_B which will produce that minimal value of T.

In another, the subunits were initially classified into quadrants by using $(B(k) - q/2, T(k) - q/2)$ as Cartesian coordinates (see Fig. 4.2). Subunits in Quadrant I have the

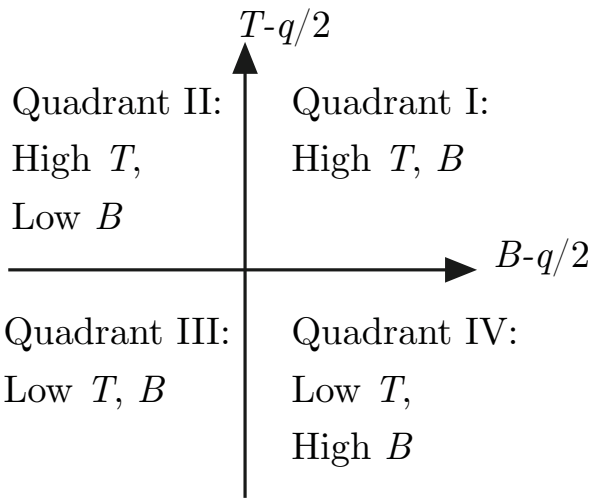


Figure 4.2. Segmentation of S in two-step method.

largest values which are hardest to fit, while those in Quadrant III have the smallest values which are easier to fit. Quadrants II and IV contain subunits with one "good" and one "bad" measurement.

Let S_{I} be the set of all subunits in Quadrant I, etc. Then the algorithm proceeded as follows:

1. Compute the distance between all pairs of subunits in Quadrants I and III:

$$d(k_{\mathrm{I}}, k_{\mathrm{III}}) = \sqrt{[T(k_{\mathrm{I}}) - T(k_{\mathrm{III}})]^2 + [B(k_{\mathrm{I}}) - B(k_{\mathrm{III}})]^2} \quad \forall\, k_{\mathrm{I}} \in S_{\mathrm{I}},\, k_{\mathrm{III}} \in S_{\mathrm{III}}. \tag{4.4}$$

2. If binding between the two subunits corresponding to the largest value of d is allowed, then join those two subunits and remove them from the pool for this phase (so this algorithm works only with the one-size protocol).
3. Repeat step #2 for the next-largest value of d until all possible matchings have been made.
4. Then repeat steps #1–#3 for Quadrants II and IV. Note that in this case it is most likely that subunits in Quadrant II will stack on top of subunits in Quadrant IV, which would produce subunits for the next phase in Quadrant I.
5. After steps #1–#4, there may still be unused subunits in each quadrant. So then repeat steps #1–#3 for the remaining pairs of quadrants in this order: I–IV, I–II, II–III, III–IV.

Section 5: Results without Scission

When $s = 8$, the one-size phased approach will find columns of the proper size without scission. Hence we present results from codes using this algorithm before launching into a discussion of the scission methods.

Table 1. Comparison of algorithms for various bins, $q = 400$.

Bin	Description
1	Top step only worked best.
2	All methods the same.
3	Top step, then bottom step worked worst.
4	Bottom step only worked best.
5	Top step only worked best.
6	Top step, then bottom step (or vice versa) worked best.
7	Top step only worked best.
8	Top step, then bottom step or top step only worked best.
9	Top step only or bottom step only worked best.

In the first set of codes, each pool was a separate bin. The assembly protocol was the single-step algorithm outlined in §4. All four permutations of the top and bottom steps were used, since the code ran very quickly. Depending on the bin, different permutations worked better (see Tab. 1). Due to an oversight, none of the deterministic algorithms analyzed bin 0.

In Fig. 5.1 we plot the percentage of delayed stacks as a function of the tolerance for each bin. Note that at the default level of $q = 400$, the number of delayed stacks varies widely by bin, with bins 7 and 9 having a 30% delay rate, while bin 5 has less than a 2% delay rate. (The percentage of delayed stacks is somewhat misleading, since several of the bin sizes were quite small; some discussion of this is contained in §8.) Note that except for bins 7 and 9, the algorithm beats the desired standard of 15% delayed stacks.

In general, the delay rate decays as a function of increasing tolerance. There is one exception for bin #6 with the tolerance equal to 410. This would be due to some anomalous combination of the assembly algorithm and the data. With the tolerance increased to 405, the algorithm beats the 15% standard for every bin. The default tolerance of $q = 400$ was arrived at heuristically, given the distributions in (4.1). The results in Fig. 5.1 (and similar subsequent results) show that increasing the tolerance by only a small amount (just over 1%) can substantially reduce the number of delayed stacks. Hence it is worthwhile for Bloom to investigate how much the tolerance can be reduced without degrading device performance.

One group also used the two-step algorithm described in §4 along with the phased one-size approach. The results are shown in Fig. 5.2. Note that in many cases the results compare favorably to the single-step approach.

In all of the previous discussion, we have ignored the problem of forming columns

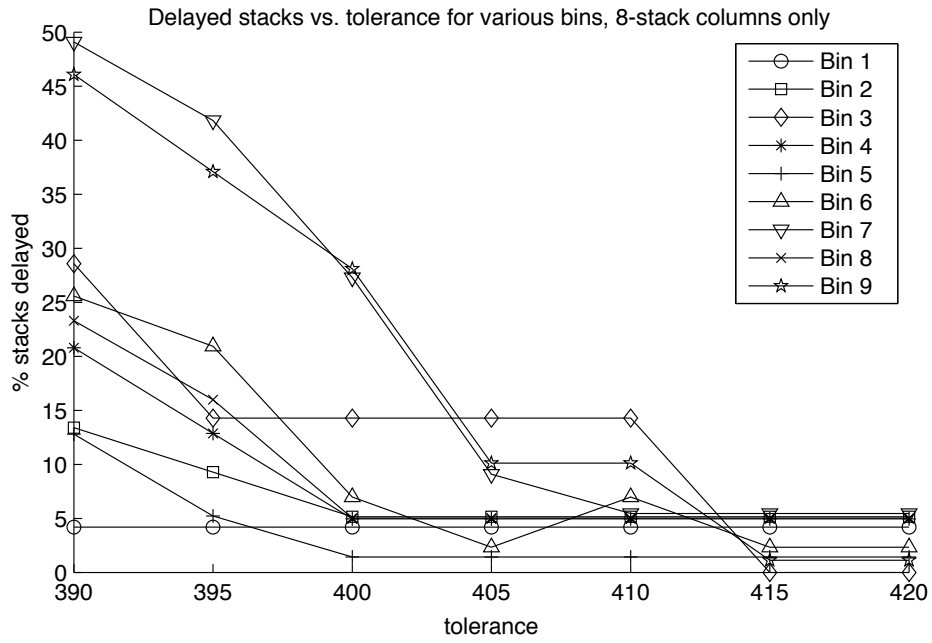


Figure 5.1. Delayed stacks as a function of tolerance for 8-stack single-bin assembly, one-step algorithm, one-size protocol.

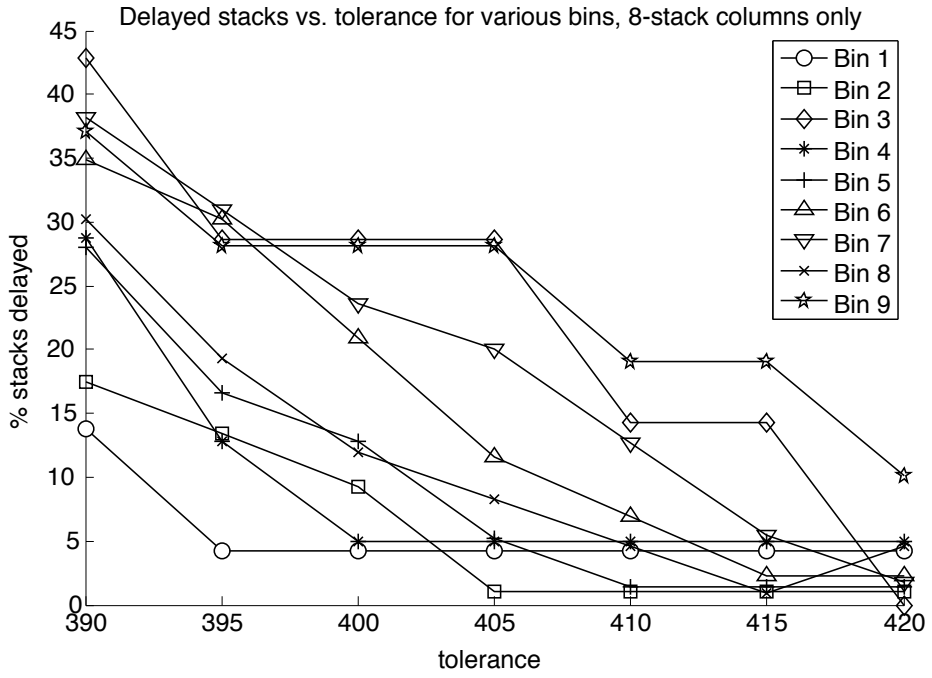


Figure 5.2. Delayed stacks as a function of tolerance for eight-column stacks, two-step algorithm, one-size protocol, single bins used.

into boxes. Unless the number of columns produced is a multiple of c, there will be spare columns in inventory until new shipments of ICs arrive. Previously, we assumed that this wouldn't be a problem if we could improve the assembly efficiency enough of single-bin columns. However, this is exactly why the additional categories were specified in assembly rule #4—to use up spare columns.

Therefore, as a next step we tried to assemble additional 8-stack columns as follows:

1. We used the single-step algorithm to create 4-stack subunits. We did this separately for two bins with adjacent numbers.
2. We then combined all the 4-stack subunits from both bins into a single pool and assembled them into 8-stack units following assembly rule #4(a), (b) (namely, that in a mixed-unit column, the higher-value bin is placed on top).

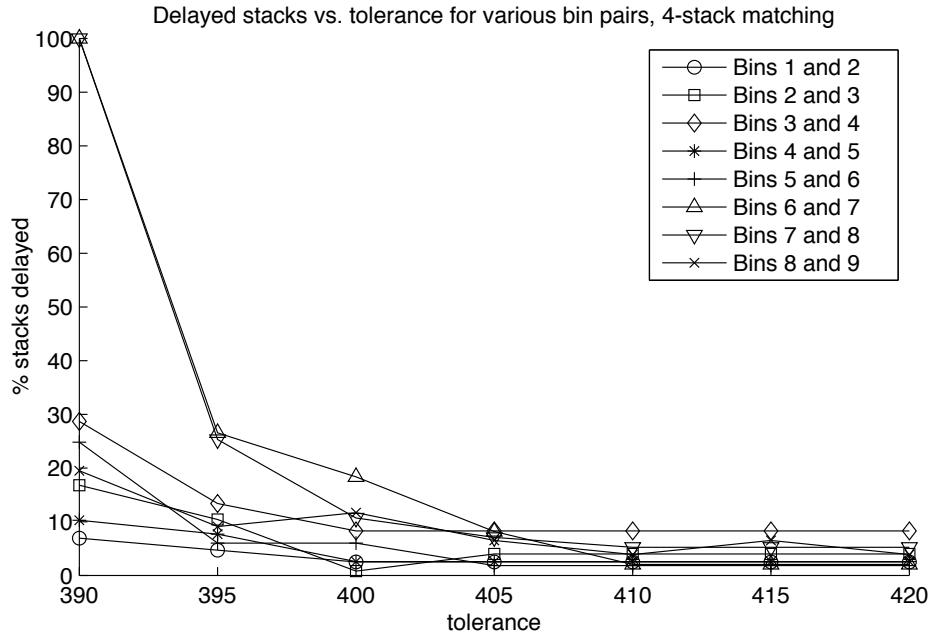


Figure 5.3. Delayed stacks as a function of tolerance for 4-stack assembly procedure.

The results are shown in Fig. 5.3. Once again, the percentage of delayed stacks is small. However, only occasionally did we get more than $c = 8$ columns of mixed type, which would be the goal in order to assemble more boxes.

Section 6: Results with Scission

If s is not a multiple of two, then the one-size assembly methods will not naturally build columns unless the subunits are broken into different sizes. Similarly, in the all-size assembly methods, it is probable that subunits with more than s stacks can be built. Hence it is necessary to derive *scission protocols* that detail how to break apart assembled subunits.

Suppose that a *superunit* has been formed of length $s + s_{\mathrm{e}}$, where the "e" denotes "extra." The protocols had to address two questions:

1. **Size of subunits**. There are two ways that the s_{e} extra stacks were placed back into the pool:
   (a) The subunit of length s_{e} is retained as a single subunit. (Note that $s_{\mathrm{e}} < s$, since otherwise a full column would have been removed at an earlier step.)
   (b) The subunit of length s_{e} is broken down into s_{e} separate stacks.
2. **Scission location**. One can remove the subunit from either the top or the bottom of the superunit. There are several different ways to make the decision about which to choose:
   (a) (Used with both options in #1.) The stacks are always removed from the top of the superunit.
   (b) (Used with both options in #1.) The stacks are always removed from the bottom of the superunit.
   (c) (Used with #1(a).) When the subunit is removed as a single piece, it will have the following end values:

   $\{B_1, T_{s_{\mathrm{e}}}\}$ (removed from bottom) or $\{B_{s+1}, T_{s+s_{\mathrm{e}}}\}$ (removed from top).

   Remove that subunit which has the smallest end value, on the hypothesis that this will be easiest to join together in a later step.
   (d) (Used with #1(b).) In this case, there is a candidate stack for removal at the top and bottom of the superunit. For each, compute the following quantity:

$$d_j = \sqrt{\left[\frac{B_j - \bar{B}}{\sigma(B)}\right]^2 + \left[\frac{T_j - \bar{T}}{\sigma(T)}\right]^2}, \tag{6.1}$$

   where $\bar{B}$ and $\sigma(B)$ are the mean and standard deviation of B, respectively, and similarly for T. Hence d as described here measures (in a normalized sense) how extreme the values of B_j and T_j are compared to their distributions. For example, at the first step one would compute d_1 (bottom) and $d_{s+s_{\mathrm{e}}}$ (top). Remove that stack which has the smaller value of d_j, on the hypothesis that this stack will be easier to join together in a later step (as its end values are closer to the mean). Repeat this step for each of the s_{e} stacks that must be removed.

We present the results from two groups that attempted scission. As noted above, when s is not a power of two, the one-size algorithm must be modified by a scission step. In order to compute the case $s = 10$, the following approach was tried. It is not the most elegant approach, but it could be easily implemented with the existing code.

1. Use the normal one-step algorithm to form columns of size 16. The unused subunits from the final step will be eight-stack columns; these will be saved as currently the factory produces both 8- and 10-stack columns.
2. Remove six stacks as a single subunit from one end of each of the 16-stack columns using location algorithm 2(c). This will make columns of size 10.
3. Create a new pool with the six-stack subunits. Match these to form ten-stack columns, again by trimming two stacks from the results using location algorithm 2(c).

The advantage of this approach is that one is always working with a population of similarly-sized stacks, so the one-size algorithm is applicable. However, there are two main drawbacks. By constructing columns larger than needed, one reduces the number of subunits available for assembly. Hence the number of additional combinations is small. Thus, on many trials the number of 10-stack columns was only one or two more than the number of 16-stack columns.

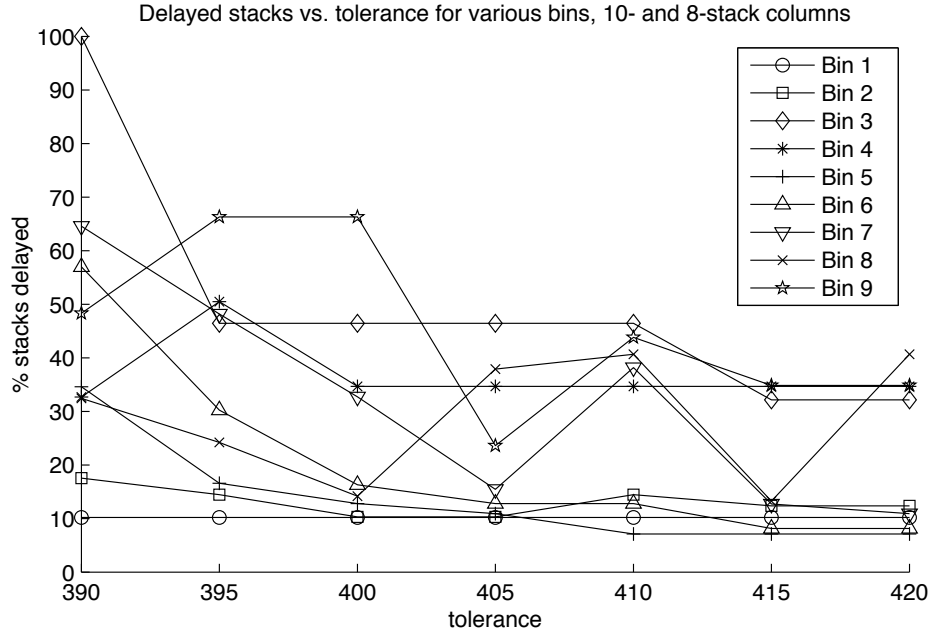


Figure 6.1. Delayed stacks as a function of tolerance for 16-stack to 10-stack one-size algorithm.

Moreover, there is an inherent lower bound on the delayed stacks in this method. Suppose the first step of the method works at maximal efficiency. Then all the stacks would initially be put into 16-stack columns. For every two 16-stack columns, a continued perfect matching would create three 10-stack columns and two delayed stacks, for a lower bound of 1/16. The inferiority of the results are shown in Fig. 6.1.

Another group implemented the all-size protocol, which necessarily requires scission. In the end the code written by this group compared the results of 28 possible assembly

protocols, returning the results from the best. These protocols were:

- For assembly, use either the top step or bottom step exclusively. For scission, use all four combinations of #1 and #2(a),(b), as well as the combination of #1(b) with #2(d). This yields ten methods.
- For assembly, use one of the two combination methods (top-bottom or bottom-top). For scission, use two location methods in order. #2(a) and #2(b) can be used repeatedly or alternately, so that yields eight scission methods for each assembly method. A ninth scission method is given by using #2(d) repeatedly with #1(b). Using these nine scission methods with both assembly methods yields an additional eighteen combinations.

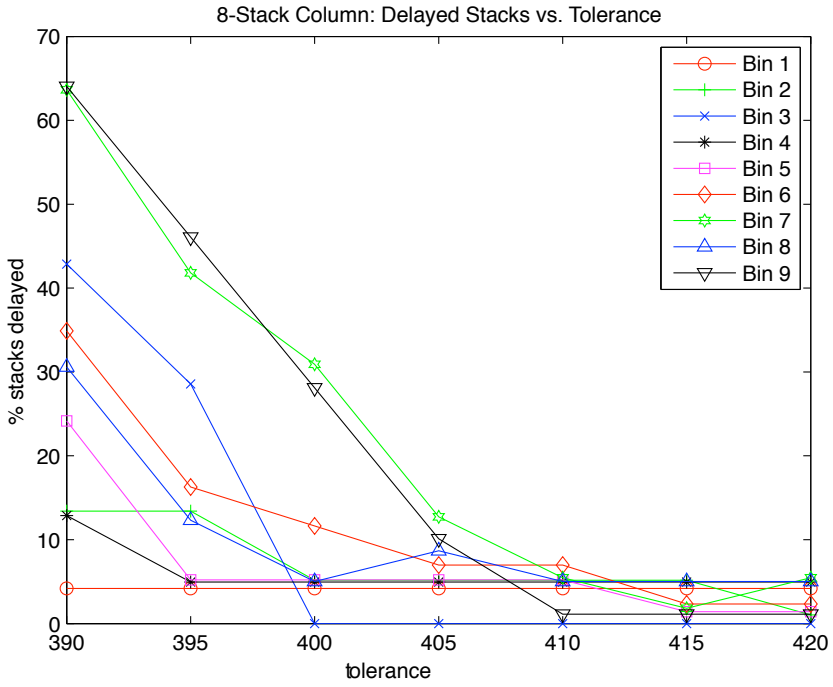


Figure 6.2. Delayed stacks as a function of tolerance for 8-stack all-size algorithm.

Running the Matlab code with all 28 combinations for all the bins took about 45 seconds on a standard laptop; the results for 8-stack columns are shown in Fig. 6.2. Note that the results compare favorably with those presented in Figs. 5.1 and 5.2 for the same 8-stack column assembly.

Moreover, this algorithm is flexible enough to handle columns with any value of s. Results for $s = 10$ are shown in Fig. 6.3. Note that the results are clearly superior to those shown in Fig. 6.1.

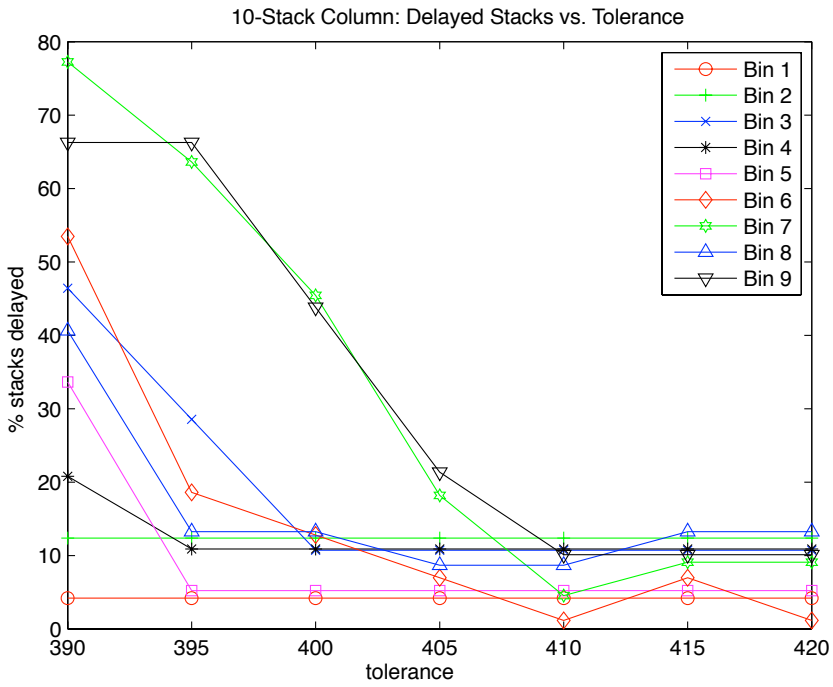


Figure 6.3. Delayed stacks as a function of tolerance for 10-stack all-size algorithm.

Section 7: Linear Programming Formulation

Our problem can also be expressed in the linear programming context. Consider a pool of N subunits. Define the *feasibility matrix* $F \in \mathcal{R}^{N \times N}$ as follows:

$$f_{ik} = \begin{cases} 1, & \text{if } i \neq k \text{ and } T(i) + B(k) < q, \\ 0, & \text{else.} \end{cases}$$

In other words, $f_{ik} = 1$ if it is *feasible* for subunit i to be stacked under subunit k. We track whether that *actually occurs* through the variable x_{ik}:

$$x_{ik} = \begin{cases} 1, & \text{if subunit } i \text{ is stacked under subunit } k, \\ 0, & \text{else.} \end{cases}$$

Then the goal is to maximize

$$H = \sum_{i,k} x_{ik} \tag{7.1}$$

given the following constraints. First, binding can occur only if it is feasible, so

$$x_{ik} \leq f_{ik} \text{ for all } i, k. \tag{7.2a}$$

Moreover, subunit i can bind with at most one other element, so we have

$$\sum_{k} x_{ik} + x_{ki} \leq 1 \text{ for all } i. \tag{7.2b}$$

Here the first sum counts bindings where i is on the bottom, while the second sum counts bindings where i is on the top.

Here the x_{ik} are either 0 or 1, so this is really an integer programming problem. It is quite similar to the marriage problem (Strang, p. 426). However, in some cases (network flows), it can be shown that the solution to the linear programming problem is always integral (Franklin, p. 139). Though we are not sure if that holds in this case, we used the Matlab binary optimization algorithm `bintprog`, which always yields binary results.

In practice, the number of constraints becomes large very quickly. Equation (7.2a) provides N^2 conditions, while (7.2b) provides $2N$ conditions. The Matlab optimization ran on a laptop with $N = 167$, but it took about 5 minutes. (The other algorithms mentioned in this report were much faster.) Moreover, we found that when we used an odd number of units (thus guaranteeing that there would not be complete matching), the linear programming code took longer than when we used an even number of units.

The group that did the linear programming approach used the one-size method. In other words, they solved the system to make pairs out of individual stacks, then ran it again to make quartets, etc. However, this is not necessary, as the size of the individual

subunits doesn't figure into the analysis. So it could be used to assemble stacks of any size.

There was some discussion about using this approach to join more than one subunit together at once. In particular, if we define

$$x_{ikl} = \begin{cases} 1, & \text{if subunit } i \text{ is stacked under subunit } k \text{ under subunit } l, \\ 0, & \text{else,} \end{cases}$$

then the goal is to maximize

$$H = \sum_{i,k,l} x_{ikl}. \tag{7.3}$$

The binding constraint then becomes

$$x_{ikl} \leq f_{ik} f_{kl} \text{ for all } i, k, l, \tag{7.4a}$$

since both merges have to be allowable. In addition, the single-binding rule becomes

$$\sum_{k,l} x_{ikl} + x_{kil} + x_{kli} \leq 1 \text{ for all } i. \tag{7.4b}$$

The problem is that we now have $O(N^3)$ constraints from (7.4a), which is quite large, and will only get larger as production ramps up and bins become larger. In general, to combine M stacks at once would take $O(N^M)$ constraints.

When solving a linear programming problem, one starts from a feasible solution and works around the edge of the simplex until an optimal solution is found. Hence permuting the data would still lead to the same number of pairings, but the pairings would be different. It was quickly shown with a small set of eight stacks (chosen with pathological B and T values) that with the stacks ordered one way, a column of eight could be formed, while with the stacks ordered another way, the best one could do was four pairs.

Hence there are cases where, though the linear programming method finds an optimal solution for the first phase, the resulting pairs are totally unsuited to be assembled in the next phase. Thus the linear programming method would do worse than the previously discussed two-step algorithms, which attempted to match extremal end values to maximize the number of stacks which could be feasibly assembled in the next phase.

This anomaly motivated the consideration of extending the two-step method to the linear programming problem. For instance, suppose that the condition in (7.1) were replaced by maximizing

$$H_B = \sum_{i,k} [q - B(i)] x_{ik}. \tag{7.5a}$$

In other words, the value of a pairing is increased if it has a lower value of B to take into the next step. When a subgroup tried this approach, they quickly found that $q - B_i$ was so large that the weighting coefficients dominated the function H_B, leading to fewer matchings. It was then suggested to try

$$H_{BT} = \sum_{i,k} \left[1 - \frac{B(i)}{q}\right] \left[1 - \frac{T(k)}{q}\right] x_{ik}. \tag{7.5b}$$

This then reduced the weightings to the interval $[0, 1]$ while also including the effect of the T value to be carried into the next round. Using this new value did not seem to affect the results appreciably.

Another idea suggested was to express the problem as a variant of the *optimal assignment problem* (Franklin, p. 145). In this case, one removes the constraint (7.2a) that grows quadratically with size. By doing this, you basically force a complete matching of the items. How then, does one enforce the condition that you can't have a matching if the curvatures exceed the tolerance?

Replace H with

$$H_F = \sum_{i,k} f_{ik} x_{ik}, \tag{7.6}$$

where f_{ik} is related to the feasibility matrix:

$$f_{ik} = \begin{cases} \left[1 - \dfrac{B(i)}{q}\right]\left[1 - \dfrac{T(k)}{q}\right], \text{ if } i \neq k \text{ and } T(i) + B(k) < q, \\ -999, \text{ else.} \end{cases}$$

Note this gives a small bonus for matching up values with good values for the next step (the two-step part), while giving a significant penalty when the algorithm makes a matching that isn't allowed physically. The idea was that this penalty would minimize the unphysical matchings, which could be classified as delayed stacks in postprocessing.

Unfortunately, none of the linear programming approaches reduced the delayed stacks nearly as much as the algorithms in the previous sections, and none came close to the 15% delay threshold.

Section 8: Random Assembly

One method of finding possible arrangements for columns (optimized or not) is to arrange the stacks randomly. To accomplish this, a column is formed one stack at a time. The code created to form a column begins by grouping all possible stacks that may be placed at the top of the stack. Note that this does not include stacks with anomalous electrical properties. A stack from this group is selected at random. Next, a group is made of all stacks that may be placed below this top stack following the criteria in (2.2) and (2.3). Of the possible stacks that match these criteria, one is selected at random. Another group is created of all stacks that satisfy the conditions to be in the third position, and the stacking is repeated until the column is constructed. Following this pattern, the algorithm essentially "draws names out of a hat" of all acceptable stacks for each position in sequence.

Once a column has been formed, it is removed from the pool, and the entire assembly process is repeated until all possible columns have been formed. Because this algorithm has no restrictions on the arrangement of stacks other than meeting the established criteria, the algorithm could list all possible arrangements for each bin value, given enough time. Thus the only restriction to this code finding the exact optimal arrangement is runtime. This is different from the other codes, which require only a limited number of time to find a near-optimal solution but by their design they may bypass a better arrangement that does not satisfy the greedy algorithm.

The main argument behind looking for arrangements randomly relates to the expectation of frequently occurring optimal solutions. In all but one case, there will never be exactly one optimal solution. For example, a bin with 64 stacks has the possibility of creating exactly 8 columns. However, since the order of the 8 columns does not matter, there are actually 8! different arrangements that are optimal. Furthermore, if any two stacks of these 64 can switch places, or if any one difference at all can occur, then there are another 8! different arrangements of these columns that are also optimal for each of the differences. While this may still be a small fraction of the 64! total possible arrangements, one must remember that only a fraction of the 64! will actually be formed, because the stacks may only be arranged to fit the criteria set in place. With this in mind one may hypothesize that there is a significant probability of a feasible solution being optimal. Hence attempting to find an optimal or near-optimal arrangement randomly carries far better odds than originally expected.

Of course, time is restricted and so a random assembly will not assuredly find the optimal solution in the allotted time. In consultation with the industry representative, we determined that the maximum amount of time the random assembly code would be allowed to run was 63 hours (Friday at 5 pm to Monday at 8 am). A better run time would be 15 hours (5 pm one day to 8 am the next). This allows employees of Bloom Energy to run the code for a relatively large amount of time without tying up computing power during business hours. The code was written in the Visual Basic component of Microsoft Excel, which is more widely known among Bloom Energy employees than other computing systems like Matlab (in which the other algorithms were written).

With these considerations, randomly assembling columns stands as a good way to measure the effectiveness of the other greedy algorithms. Thus, this method of arrangement will work as a control group in a scientific study: as a way to ask "how much better is this algorithm than randomly arranging the stacks?" Knowing that another algorithm outperforms random assembly suggests that algorithm might be a step in the right direction to finding the optimal solution. Likewise, if randomly assembling stacks creates fewer delayed stacks than another algorithm, then that algorithm likely creates suboptimal solutions.

To obtain the results, the same data was used by random assembly as by the other greedy algorithms. Since time is the only issue for random assembly, the total computation time was kept below the maximum and computers of varying performances were used. 5000 simulations were run over a total computing time of roughly 60 hours on 10 different computers. A key benefit to being restricted only by computation time is that the actual elapsed time may be reduced by simply running the code on multiple machines simultaneously. The random number generator was confirmed not to repeat when the code was restarted so the use of multiple computers and multiple restarts is possible.

It is important to reiterate that the argument here is not that there are only a few possible arrangements in total (in this case, using random assembly to find the optimal solution is akin to buying more lottery tickets to improve one's chances of winning). The argument is that, of the arrangements found randomly, highly efficient arrangements will be present and will occur multiple times, suggesting that finding near-optimal solutions is likely even without an algorithm designed to assemble stacks efficiently.

The figures below show the number of times arrangements with different percentage of delayed stacks were found. In each case, columns of length 8 were made, so the total stacks used will always be divisible by 8. The figures are labeled in terms of how many stacks were used to make columns. All of the greedy algorithms were tested by counting how many columns were created instead of the number of boxes created; thus the random assembly was measured in the same way.

Figs. 8.1 and 8.2 give an idea of how often an arrangement using a given number of stacks will occur. After 5000 iterations, some delay percentages still do not occur and suggest their rarity (or impossibility). For instance, consider bin 0, which with 37 stacks would have theoretical upper bound of four columns. However, observation of the stacks in this bin shows that the top and bottom values of many of these stacks are over $q/2 = 200$ each, and so assembling more than one column may well be impossible. However, a single column was found many times and stands as a perfect example of the occurrence of optimal solutions. With bin 0 specifically, we can expect each algorithm to create a single column and should take notice if an algorithm does not. Likewise, we should be wary if any greedy algorithm creates an arrangement worse than the worst found randomly.

To summarize the effectiveness of the greedy algorithms, their efficiencies (in terms of number of columns created) were compared to the best solution found by random assembly. Note that random assembly underperformed for some bins, and outperformed for others. The results are summarized in Fig. 8.3. Here if the plot has a positive y-value, the greedy algorithm outperformed the random algorithm. As discussed in §5, the greedy algorithms did not analyze bin 0, so it does not appear in the analysis. Because of the long run time

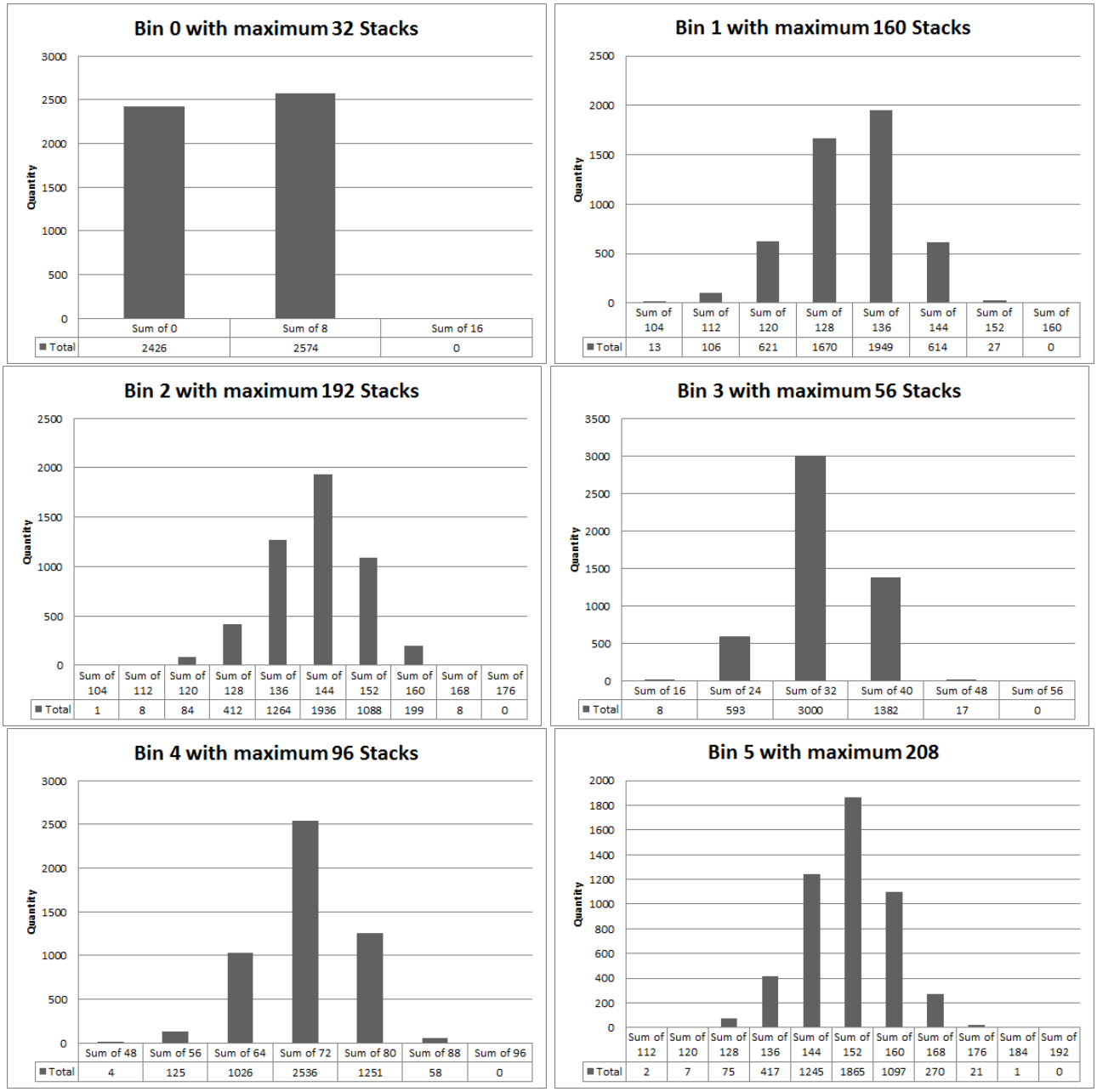


Figure 8.1. Results from random assembly for bins 0–5, $q = 400$. The horizontal axis corresponds to number of columns formed.

involved, we were able to test only the default tolerance value of $q = 400$ against the greedy algorithms.

Note that for bin 9 each greedy algorithm underperformed random assembly. This may be an example of a case where the optimal solution does not follow the generally believed rules of an efficient arrangement.

More generally, the results show that arrangements from the random assembly method were within a column or two from those generated by the greedy algorithms. (The difference seems so large from bin 3 because the number of stacks there is so small; a one-column

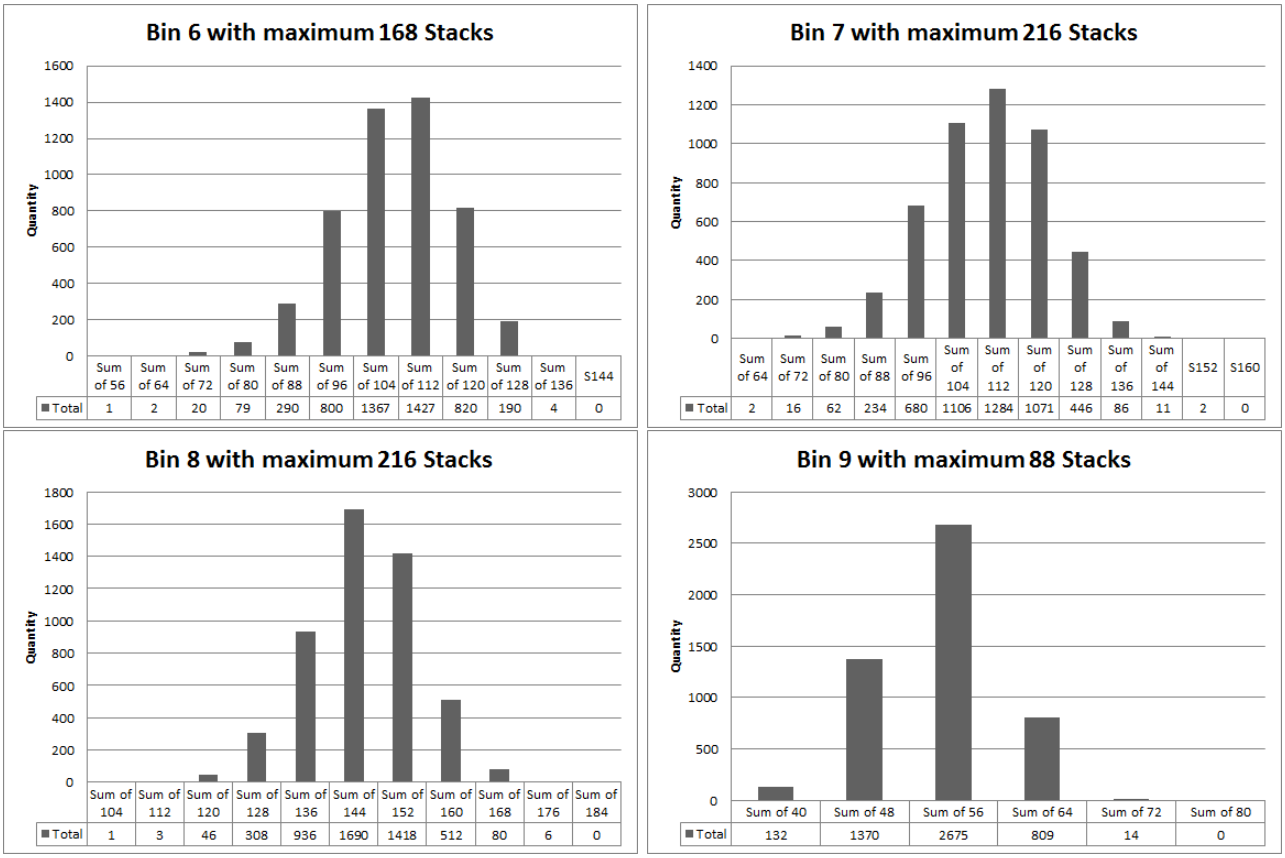


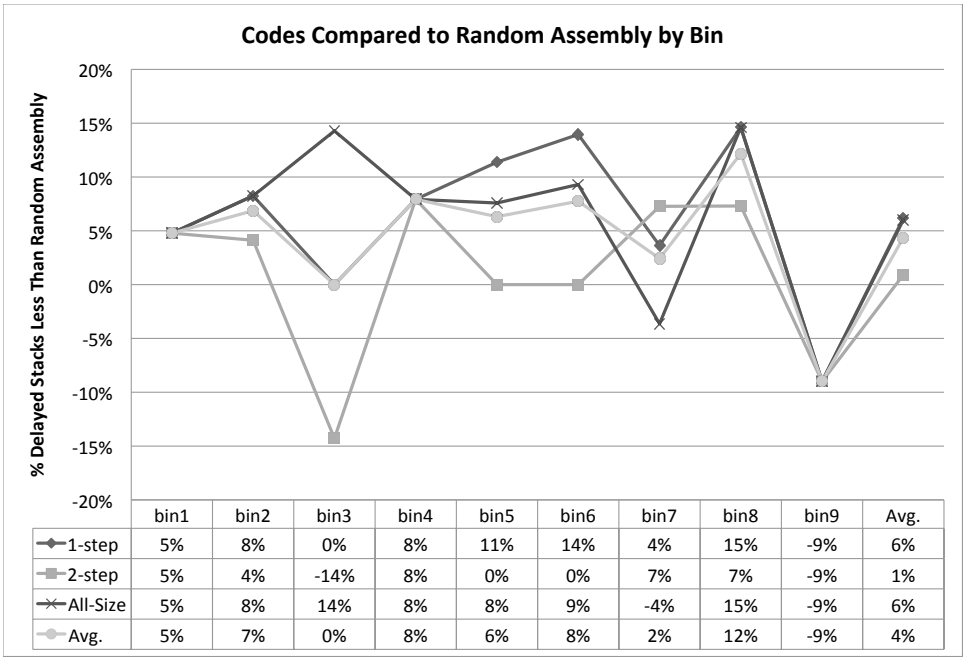
Figure 8.2. Results from random assembly for bins 6–9, $q = 400$. The horizontal axis corresponds to number of columns formed.

difference causes 14% delayed stacks.) Given that no training would be required to implement the random assembly's Visual Basic code (and that the optimality would likely increase with more processors), these results might be good enough for Bloom to pursue the random assembly strategy.

Because of time constraints, we did not analyze the case where columns could stacks from adjacent bins, as illustrated in Fig. 5.3. However, the current code for random assembly can be easily modified to build as many columns as possible using assembly rule 4(a) (where all stacks are of the same bin) and then build as many columns possible using assembly rule 4(b) (where stacks come from adjacent bins). Refining the code in such ways will give a better idea of how often low-delay arrangements occur.

Of course, this could also be done to find an optimal solution to the full problem. In particular, the code for finding random arrangements can be easily modified to include additional constraints (such as the A2 value) or fewer constraints (such as relaxing the same-bin requirement). An important note is that since additional constraints will decrease the number of possible arrangements, including them will allow for the code to find a larger percentage of the total combinations in less time.

Further enhancements to the random assembly code should include the constraint that 8 columns must be identical to be used in a single box, as discussed in §2. Thus, finding 1, 2, or 7 columns is not a valuable achievement, as the stacks used in those columns may



	bin1	bin2	bin3	bin4	bin5	bin6	bin7	bin8	bin9	Avg.
1-step	5%	8%	0%	8%	11%	14%	4%	15%	-9%	6%
2-step	5%	4%	-14%	8%	0%	0%	7%	7%	-9%	1%
All-Size	5%	8%	14%	8%	8%	9%	-4%	15%	-9%	6%
Avg.	5%	7%	0%	8%	6%	8%	2%	12%	-9%	4%

Figure 8.3. Comparison of random assembly to greedy algorithms. Here if the data point corresponding to a greedy algorithm has a positive y-value, it outperformed the random algorithm. "1-step" is the single-step assembly illustrated in Fig. 5.1. "2-step" is the two-step algorithm illustrated in Fig. 5.2. "All-size" is the all-size assembly illustrated in Fig. 6.2: however, the comparison was made before all 28 procedures in the all-size algorithm were implemented.

help create a complete set of 8 columns elsewhere. Again, updating the current code to allow only groups of 8 columns to be produced is fairly simple and is already in place, but was not used for the control test.

Section 9: Postprocessing

Once the assembly process is complete, we can try some postprocessing measures in order to improve our results further. If we have not used the full pool of available stacks as our initial S, we can augment S with additional stacks from the pool in order to continue the algorithm. If the pool is exhausted, one can break any unused subunit with more than one stack back into its component stacks to see if these smaller stacks can be reassembled into columns. These ideas were considered, but not implemented.

However, one approach that the group did implement was the use of a genetic algorithm. These sorts of algorithms take ideas from genetics to try to improve the results from one of our previous algorithms. We define an *arrangement* U to be the set of columns and unused subunits (*across all bins*) generated by one of our previous algorithms. For the purposes of the genetic algorithm, the unused subunits should all be in individual stacks.

Assume moreover that we are given a set of M different arrangements initially (this is called the first *generation*). Note that, as currently implemented:

- the simplest one-size algorithm generates four different arrangements
- the implemented all-size algorithm generates 28 different arrangements
- the random-assembly algorithm could generate as many as desired
- in principle, the linear programming algorithm could also generate as many as desired (by permuting the data)

The algorithm proceeds as follows:

1. Compute $P(U_i)$, the percentage of stacks used in columns in arrangement U_i, and then compute a *reproduction probability* p_{r} as follows:

$$p_{\mathrm{r}}(U_i) = \frac{P(U_i)}{\sum_i P(U_i)}.$$

   In other words, the probability that a particular arrangement will be retained for subsequent generations is proportional to the percentage of stacks it assembles into columns.
2. **Reproduction step**. To compute the population for the next generation, select M arrangements from the previous generation (with replacement), where the probability distribution is given by p_{r}. Hence it is likely that there will be multiple copies of certain arrangements, and it is likely that those arrangements will be those that did the best at reducing the number of delayed stacks.
3. **Mutation step**. With some probability p_μ (the *mutation probability parameter*), make a small change in an existing arrangement by exchanging one stack from a column for an unused stack that satisfies the constraints. In general, p_μ is kept small since useful information (*i.e.*, efficient arrangements) may be lost in the mutation.
4. **Additional assembly**. Once the mutation step is complete, examine all the unused stacks to see if they can be assembled into columns using whatever assembly algorithm is being used.

5. **Crossover step**. With some probability p_c (the *crossover probability parameter*), choose U_i for the crossover step. The crossover step requires two arrangements, so if U_i is chosen, pick another arrangement from the remaining $M-1$ with equal probability. Note that the arrangement is the total set of assemblies for all bins. Hence each arrangement U_i will have *subarrangements* $U_{i,j}$ corresponding to bin j.
   Without loss of generality, assume that U_1 and U_2 are chosen for the crossover step. We now wish to construct new arrangements (U_1^*, U_2^*) for the next generation. To do so, we will exchange $U_{1,j}$ with $U_{2,j}$ (the *crossover*) with probability 1/2.
   With the reshuffling of subarrangements, there is now a new pool of unused stacks from which columns can be assembled. However, for assembly purposes the pool from the new arrangements is distinct (and hence useful) **ONLY** in the case where columns can be assembled from stacks from adjacent bins. If instead we require that each column must be made from stacks from the same bin, this step simply achieves a reshuffle of the $U_{i,j}$, which would then be fed into the reproduction and mutation steps.
   In contrast to the mutation step, in the crossover step efficient subarrangements are retained, though they may be exchanged between arrangements. Hence the range of useful p_c is larger.
6. Now a new generation has been created, so repeat the process.

The algorithm also has the capability to assemble boxes, though those results are not presented here.

The results of the genetic algorithm are shown in Fig. 9.1. It shows the delayed-stack percentage of the best solution after ten generations for $M = 30$. In this case, the initial arrangements were generated by a random assembly algorithm quite similar to that in §8. Since there were only 30 arrangements generated (rather than the 5000 of §8), it is more unlikely that near-optimal solutions were fed to the genetic algorithm as generation 0. Hence it is no surprise that the results in Fig. 9.1 are worse than those from previous sections.

Due to their nature, genetic algorithms are highly sensitive to the quality of the first generation, unless one is willing to invest significant time to allow the mutations to produce better results. Hence if the arrangements from previous sections had been used as the first generation for the genetic algorithm, obviously the results would be better than those in Fig. 9.1.

The algorithm is also highly sensitive to the parameters used: M, p_μ, and p_c. The success of the algorithm will depend on those parameters, and the best performance will be problem-dependent. (More details about genetic algorithms can be found in Michalewicz (1999).) So some experimentation would be required to implement this efficiently for the fuel-cell assembly problem.

Nevertheless, the genetic algorithm does have several advantages. Due to its randomness, it can explore various areas of arrangement space that the greedy algorithms might miss. However, it still has a deterministic selection mechanism that focuses its efforts on regions with high $P(U)$.

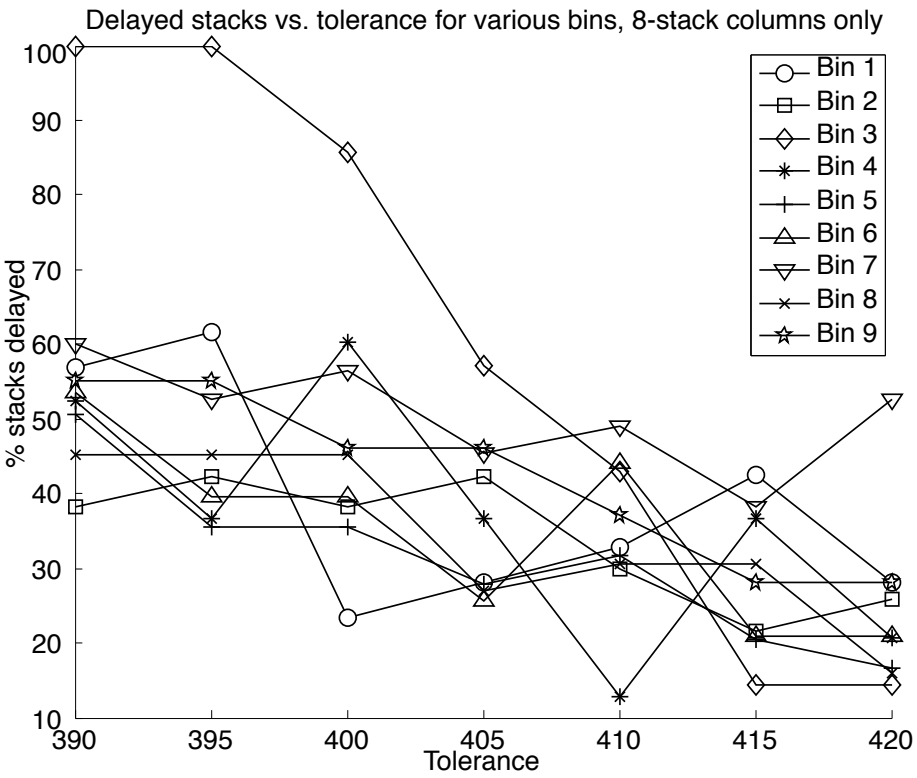


Figure 9.1. Delayed stacks as a function of tolerance for eight-column stacks, random assembly, genetic algorithm, ten generations. Here $M = 30$, $p_\mu = 0.01$, and $p_\mathrm{c} = 0.05$.

Section 10: Conclusions and Further Research

Perhaps the main surprise from the week was that the various heuristic algorithms for building columns from stacks seemed to work so well, given their simplicity. Most of the algorithms were greedy; that is, they built up columns iteratively by trying to make a locally optimal choice (chosen differently for the different strategies) at each step. It is well known that greedy algorithms will generally not find the true optimum; it is also known that the structure of some problems is such that a greedy algorithm can produce the worst possible solution! Though a hint of such an anomalous structure was seen in bin 9 (where all the greedy algorithms underperformed the random-assembly algorithm), in general the greedy algorithms provided the best solution.

The group attempted one- and two-step assembly methods using one- or multiple-size building blocks. All of the assembly methods usually came with a column or two of one another. In general, the multiple-size method worked best, as it has more flexibility in choosing subunits to assemble as compared with the one-size methods, where subunits are either matched or discarded in a series of iterative steps. The multiple-size method is also clearly superior when s is not a power of 2.

Though the greedy algorithms worked well, they were all written in Matlab, with which few Bloom employees have familiarity. The random-assembly algorithm outlined in §8 was written in Visual Basic, which is more widely known at Bloom. With the number of iterations performed for this manuscript, in general the random-assembly method did not work as well as the greedy algorithms; however, the deficiency was only a column or two. With the current number of shipments, this deficiency could probably be erased with additional processors.

The group also formulated the problem in the linear programming context, but found actual implementation infeasible. The difficulty is that as the number of components increases, the rate of increase in problem size (and hence run time) is far slower for the greedy algorithms than for the linear-programming and random-assembly formulations. Hence it seems quite possible that the greedy-algorithm strategies presented here will be more useful to Bloom in the long term, and in fact that their utility will improve as production increases.

One way to merge the two approaches is with a genetic algorithm. Using the results of a greedy algorithm as a starting point, the genetic algorithm can randomly perform changes to see if a more optimal arrangement can be found. Given that the results of the greedy algorithm were commonly within just a few columns of optimality anyway, the genetic algorithm should work well in determining whether additional columns can be formed.

One of the questions brought to the workshop by Bloom was how they might modify the assembly rules currently in use to increase efficiency. Although the group really only looked at a subset of the real problem, we were able to investigate the dependence of the

number of delayed stacks on the tolerance q in (2.2). In many cases, as might be expected, the number of delayed stacks decreased as the tolerance was increased. In particular, even small changes in q caused great decrease in the delay percentage. There were some anomalies where the number of delayed stacks increased as tolerance increased, which would seem to be a defect in the greedy algorithms that were used.

Much work remains to be done. The two most pressing issues are to include the vendor constraint, and the problem of assembling columns into boxes. One of the assembly rules that Bloom uses is to attempt to avoid mixing components from different vendors (see §2). Bloom told the group to ignore this rule for simplicity; currently only the random-assembly algorithm has the capacity to include this constraint. However, implementing this rule would seem to require a relatively straightforward modification of the greedy algorithms; effectively the existing bins of components would be subdivided by vendor, which would actually decrease the size of the optimization problem (although as noted above, performance of the greedy algorithms generally seemed to improve with larger data sets).

The problem of assembling columns into boxes has a slightly different structure than the problem of assembling stacks into columns. Stacks need to satisfy a tolerance condition with their nearest neighbors; in contrast, all columns in a box should have an average bin value which is the same or at least close in value (see §2 for details). If the columns generated are ranked by average bin value, a constraint of this nature would be easy to impose. The new wrinkle is that it might make sense to settle for a smaller number of columns, if their bin values are such that more complete boxes can be assembled. This will require an algorithm that looks ahead in some way when assembling boxes, in much the same way that the "two-step" greedy algorithm looked ahead when assembling columns.

Nomenclature

The equation number where a symbol first appears is listed, if appropriate.

A: parameter related to pore area of IC (2.1).
B: absolute value of curvature of bottom of stack.
b: index related to B value (4.3).
c: number of columns in a box.
d: distance metric, variously defined.
F: feasibility matrix for linear programming problem.
G: variable describing abnormality.
H: function to be optimized in linear programming formulation (7.1).
i: integer used to index elements.
j: integer used to index position (2.2).
k: integer used to index elements (2.3a).
$L(k)$: length of element k.
l: integer used to index elements.
M: integer, variously defined.
$\mathcal{N}$: normal distribution (4.1).
N: number of subunits in initial pool.
$P(U)$: percentage of stacks used in columns in arrangement U.
p: probability of event in genetic algorithm.
Q: dummy curvature value given to anomalous stacks (2.3a).
q: tolerance for curvature sum (2.2).
R: number of repeating units in a stack.
S: set of elements in greedy algorithm.
s: number of stacks in a column.
T: curvature of top of stack.
t: index related to T value (4.2).
U: arrangement in the genetic algorithm.
x: variable in linear programming problem.
$\sigma(\cdot)$: standard deviation of $\cdot$ (6.1).

Other Notation

c: as a subscript on p, used to indicate the crossover step.
e: as a subscript on s, used to indicate extra stacks.
r: as a subscript on p, used to indicate the reproduction step.
μ: as a subscript on p, used to indicate the mutation step.

$\bar{}$: used to indicate the mean (6.1).
$-$: as a superscript, used to indicate minimum (4.2).
$+$: as a superscript, used to indicate maximum (4.3).
$*$: as a superscript, used to indicate a crossover arrangement.

References

Franklin, J. *Methods of Mathematical Economics: Linear and Nonlinear Programming, Fixed-Point Theorems*. New York: Springer-Verlag, 1980.

Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*. New York: Springer-Verlag, 1999.

Strang, G. *Linear Algebra and Its Applications*, 3rd ed. New York: Harcourt Brace Jovanovich, 1988.