MICS JOURNAL

Mathematics-in-Industry Case Studies Journal, Volume 6, pp. 22-47 (2014)

# Improving a Fuel Cell Assembly Process

Ibrahim Diakite [*] David A. Edwards [†] Brooks Emerick [‡]
Christopher Raymond [§] Matt Zumbrum [¶] Mark J. Panaggio [‖] Angela L. Peace [**]

**Abstract.** When fuel cell modules are built from individual components, the components must be assembled according to certain rules based on manufacturing tolerances. As production increases, computer implementation of selection algorithms is essential for the speedy, efficient use of the available components. Several greedy algorithms are presented which quickly produce an assembly schedule that maximizes the number of components used from a particular inventory. These algorithms use both stepwise and one-stage approaches to the larger assembly, and some "look ahead" to the next stage in order to further maximize the number of components used. Once these approaches yield results, a genetic algorithm can be used to further optimize the production schedule. Results are presented for real-world data which compare very favorably with procedures currently in practice.

**Keywords.** fuel cells, genetic algorithms, greedy algorithms, selection algorithms, linear programming

## 1 Introduction

Fuel cells are a promising technology for environmentally friendly power generation, for (depending on the feedstock) they emit little to no greenhouse gases as waste product. Solid oxide fuel cells (SOFC) are popular in large-scale industrial applications. In a typical SOFC, hydrogen and air are introduced on either side of a cell component (see Figure 1). If pure hydrogen gas is introduced, the only waste product will be water, as shown in Figure 1. If the hydrogen gas is derived from a

[*]Department of Mathematics, University of Texas, Arlington, Arlington, TX 76019, ibrahim.diakite@mavs.uta.edu
[†]Department of Mathematical Sciences, University of Delaware, Newark, DE 19716 [corresponding author] edwards@math.udel.edu
[‡]Department of Mathematical Sciences, University of Delaware, Newark, DE 19716.emerick@math.udel.edu
[§]Department of Mathematical Sciences, University of Delaware, Newark, DE 19716.craymond@math.udel.edu
[¶]Department of Mathematical Sciences, University of Delaware, Newark, DE 19716.zumbrum@math.udel.edu
[‖]Department of Engineering Sciences and Applied Mathematics, Northwestern University, Evanston, IL 60208. markpanaggio2014@u.northwestern.edu
[**]School of Mathematical and Statistical Sciences, Arizona State University, Tempe, AZ 85287. angela.peace@asu.edu

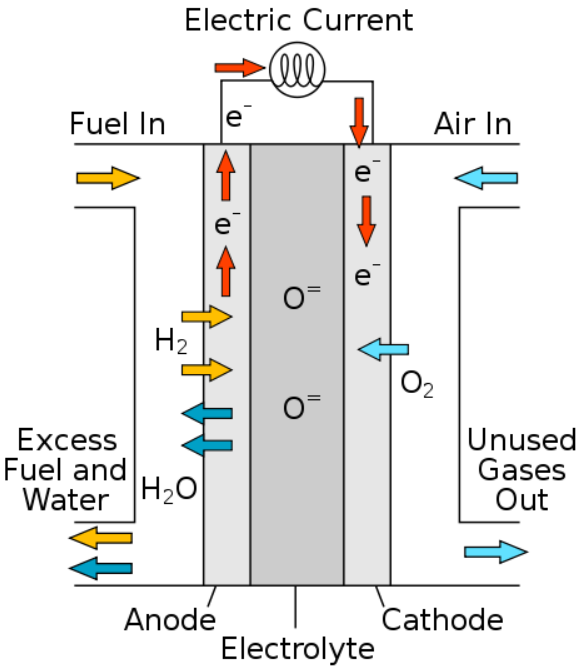


**Figure 1:** *Diagram of single solid oxide fuel cell (SOFC). In this schematic, pure hydrogen gas is used as the feedstock.*

larger hydrocarbon (such as natural gas), there will be some exhaust greenhouse gases produced, but ideally at much lower rates than to generate the same amount of electricity *via* other means.

At the cathode, oxygen in the air combines with electrons to form oxygen ions, which diffuse through the electrolyte. At the anode, these ions combine with hydrogen gas to form water. The additional electrons released in this reaction cannot easily diffuse through the electrolyte; instead they are diverted through a wire to produce electricity.

The output voltage of a single fuel cell is typically less than a volt; hence the cells must be combined (usually in series) into *stacks* which then can provide the necessary voltage. Hence what is typically called a "fuel cell" is really a module based on a series of *repeating units* (RU), as shown in Figure 2. In particular, two cells are separated by an *interconnect* (IC), which has channels to introduce feedstock and remove waste. (For more details on fuel cell technology, see [3] and [4].)

These components have physical properties which vary from unit to unit. Power modules assembled from components with similar physical properties typically have superior performance over those with a great variability. Therefore, it is critical to have a systematic approach for sorting through the thousands of components and assembling those sets which will maximize device performance.

Acceptable constraints on these variations are determined by manufacturers using computational modeling and experiments. In addition, a few components have properties which limit their

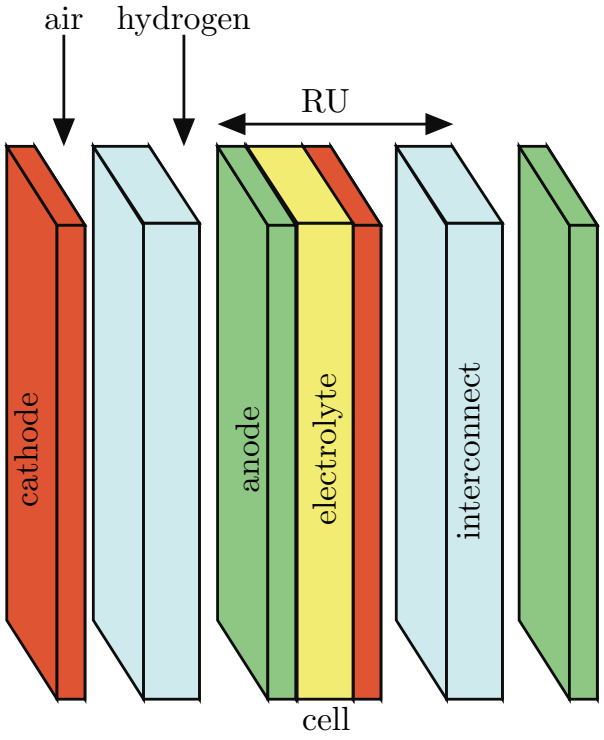


**Figure 2:** *Diagram of a repeating unit.*

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. This strategy, which is adequate for small-scale fuel-cell assembly, becomes unworkable when dealing with an inventory of tens of thousands of components. Hence for assembly on a large scale, an automated method for generating a production schedule is needed.

Manufacturers 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 immediately be assembled under the existing production schedule. Thus these components must be held in inventory until new components arrive, slowing down production and incurring associated costs.

The assembly process proceeds in stages: the basic components are assembled into stacks, the stacks are assembled into columns, and finally the columns are assembled into boxes. At each stage, manufacturers have 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, we focus on a simplified subproblem, that of assembling stacks into columns, subject to a subset of the rules actually used in practice. If this simplified problem could be effectively (*i.e.*, quickly and approximately, but to good accuracy) solved, then variations of the same idea can be used for the other assembly levels. This simplified problem of generating columns from stacks can in fact be posed as an integer programming (IP) problem; unfortunately, even this subproblem is too large to attack directly.

In this manuscript we explore several possible approaches to the problem. Most use iterative strategies to build columns. The strategies were tested on real data provided by the manufacturer. All of these iterative approaches ran quickly and produced delay rates which were often competitive with or superior to a benchmark of 15%. None of these heuristic strategies was clearly superior to all others for all sets of inputs. However, they all run very quickly and scale reasonably well with the number of stacks in the system. Hence in principle they could all be implemented, with the best result chosen for any particular set of inputs. Similar strategies may be implemented for the other assembly levels.

Section 2 of this manuscript discusses terminology and outlines the assembly rules used in practice. Section 3 discusses some of the underlying ideas common to all the heuristic assembly strategies that were implemented. In particular, we note 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 discusses the use of a genetic algorithm strategy to improve results.

## 2 Preliminaries

In order to simplify later presentation, we begin by presenting some definitions used to characterize the fuel cell technology. The largest piece of equipment actually shipped is called a *box* (see Figure 3). Each box contains $c$ *columns* where the power generation takes place (for a detailed list of all variables, see the nomenclature section at the end of this manuscript). 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.

Each stack is made up of $r$ repeating units (RUs), which are made up of an IC and a cell (see Figure 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 is identified with a value of the parameter $A$, which is related to a key characteristic of 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

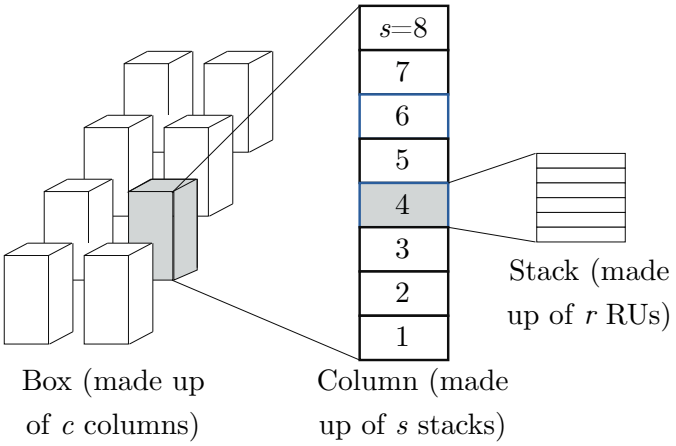


**Figure 3:** *Schematic of assembly levels illustrating an eight-stack column. This manuscript focuses on the assembly of stacks into columns.*

of all the ICs in that shipment. Thus it is impossible to know the value of $A$ for any particular IC. Depending on the value of $A$, the manufacturer can identify the shipment with a *bin*. Currently, the bins are numbered from 0 to 9 using the following rule:

$$A \in (0.14 + 0.01M, 0.15 + 0.01M) \qquad \Longrightarrow \qquad \text{shipment in bin } M. \tag{1}$$

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. Thus it is preferable 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. Therefore it is quite easy to assemble the ICs into stacks. In particular, 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 manuscript, we ignore stack assembly and treat the stack as the lowest-level building block. We then focus on the assembly of stacks into columns.

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 (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.

Some additional properties are known 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.

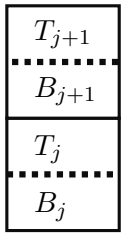


**Figure 4:** *Schematic of assembly rule #3, as shown in (2).*

2. $T$ is a positive number that characterizes certain physical properties of the top of the stack.

3. $B$ is a positive number that characterizes certain physical properties of the bottom of the stack.

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

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

where $q$ is a tolerance value (see Figure 4). Initially $q$ is taken to be 400. Equation (2) ensures that the fuel cell works properly 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 non-decreasing bin number.

As assembly proceeds, intermediate *subunits* will be produced which are not yet full columns. We define the *size* of such a subunit to be the number of stacks it contains. 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. Finally, the hundreds of columns are assembled into dozens of boxes. We assume that the column-selection process mimics the bin-selection process in rule #4.

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

$$\text{anomalous shape in stack } k \qquad \Longrightarrow \qquad T(k) = Q, \qquad Q > q. \tag{3}$$

Note that we use the argument notation $T(k)$ to refer to properties of the $k$th stack in the initial pool, while we use the subscript notation $T_j$ to refer to the stack in the $j$th position in a particular column.

With the definition in (3), we see that (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 (3) makes rule #1 redundant. Similarly, if we define

$$\text{anomalous electrical property in stack } k \qquad \Longrightarrow \qquad B(k) = Q, \tag{4}$$

then (2) will never be satisfied for any stack with an anomalous electrical property. Therefore, 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 prohibitively expensive to compute (for further discussion, see 4.3). Therefore, we largely focus on simple and fast algorithms that could yield nearly-optimal results. There are many separate facets to consider:

1. The *pool protocol*, which determines the initial set of stacks to assemble.
2. The *subunit 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 protocol*, which dictates whether assembled subunits are broken down into smaller parts.

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

# 3 Pool and Subunit Size Protocols

## 3.1 Pool Protocols

There are several ways to choose the initial set of stacks to assemble. Assembly rule #4(a) motivates the most obvious choice: use one of the pre-identified bins in (1) as the pool. Though one might

think that such a choice would be too restrictive, our results in the following sections will show that one can obtain small percentages of delayed stacks in this case.

For comparison, we also will present results from a pool protocol motivated by assembly rule #4(b). Perhaps surprisingly, the results are not particularly improved from the one-bin pool protocol, at least with the algorithms used.

There are many other more complicated ways to choose the initial pool; some of these are discussed in 8.2.

### 3.2 Subunit Size Protocols

Once the assembly process begins, we need to decide what size subunits are available for assembly at each step of the algorithm. We present results from two different approaches.

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

**All-size protocol.** In this approach, assembly takes place in one phase. Once two subunits are combined, the resulting larger subunit remains in the pool to be joined again. Hence, at any point, subunits of various sizes are available to be combined. The only time a subunit is removed from the pool is when it forms a column.

## 4 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 as follows:

$$T \sim \mathcal{N}(215, 50), \qquad B \sim \mathcal{N}(180, 70). \tag{5}$$

A typical data set (including values of $T$, $B$, and the shape and electrical properties) is included as an Excel file in the supporting documentation for this manuscript. The true data sets are proprietary.

Given the variable distributions in (5), the default tolerance of $q = 400$ in (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 $q$.

Given this tight tolerance, assembling stacks at random should lead to many delayed stacks. Hence an optimal assembly protocol would greatly improve the assembly process.

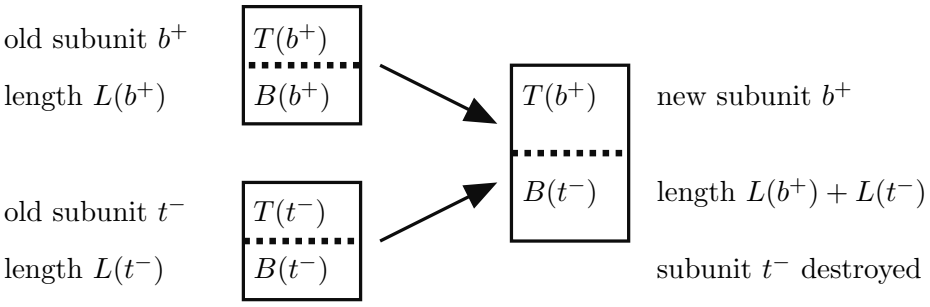


**Figure 5:** *Combining of two stacks using top method.*

### 4.1 One-Step Algorithm

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

**Top Step.** Define the subunit index $t^-$ as follows:

$$T(t^-) = \min_S T(k). \tag{6}$$

Define $S_B$ to be the set of all possible subunits that can bind with subunit $t^-$ and still satisfy the tolerance:

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

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

$$B(b^+) = \max_{S_B} B(k). \tag{7}$$

(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 Figure 5). Mathematically, we remove $t^-$ from $S$ and redefine the triplet corresponding to $b^+$ as follows:

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

By using the largest possible values of $B$ during assembly, the subunits that result have smaller values of $B$, which increases their chances of being able to be assembled into even larger subunits in the next stage. However, no explicit consideration is given to the $T$ and $B$ values of the assembled subunit. Hence we refer to these type of algorithms as *one-step algorithms*, since they optimize values only within the step at hand.

Alternatively, we may use the **Bottom Step** method, which simply switches the roles of top and bottom from the "top step" method listed above. In particular, we have

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

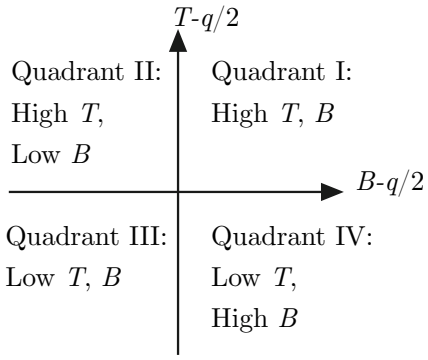


**Figure 6:** *Segmentation of $S$ in two-step method.*

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

These two methods actually provide four separate one-step assembly options. One can assemble the subunits 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, one may use all four of these methods to find the solution that uses the most subunits.

### 4.2 Two-Step Algorithm

In contrast to the one-step algorithm described above, a *two-step* method explicitly attempts to produce a set of subunits that can be optimally assembled at the next stage.

One such method replaces the definition of $b^+$ in (7) with

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

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

In another, the subunits are initially classified into quadrants by using $(B(k) - q/2, T(k) - q/2)$ as Cartesian coordinates (see Figure 6). Subunits in Quadrant I have the 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 $S_{\mathrm{I}}$ be the set of all subunits in Quadrant I, etc. The algorithm proceeds as follows:

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

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

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. 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. Repeat steps #1–#3 with the unused subunits from the remaining pairs of quadrants in this order: I–IV, I–II, II–III, III–IV.

Note that both the one- and two-step algorithms take a set of subunits and pair them up as best as possible. If the resulting larger subunits do not produce columns, the algorithms must be repeated again with the larger subunits as the base pool.

### 4.3 Integer Programming Formulation

The problem can also be expressed in the IP context; a detailed formulation is given in 9, the Appendix. For our purposes, it is sufficient to summarize the main weaknesses of the approach.

In the simplest formulation of the IP method, the goal is to maximize the number of pairings. The advantage of such a formulation is that it provides a method for searching the entire space of possible solutions. However, this systematic approach has a high computational cost. An initial pool of $N$ subunits generates $O(N^2)$ variables $x_{ik}$ (corresponding to subunit $i$ stacked under subunit $k$) and $O(N^2)$ constraints to be satisfied (corresponding to whether such a stacking is allowable under the assembly rules). Therefore, even with a single bin ($N = O(10^2)$), the time needed just to construct pairs can be substantial, especially in comparison with the other algorithms described above.

In most cases, there were many possible configurations which had the same number of pairings. Thus the additional computational time consumed by the IP method was not well spent, since the heuristic algorithms could construct the same number of pairings much more quickly.

Moreover, the heuristic methods attempted to "look ahead" and produce subunits that could be easily assembled at the next level. In contrast, by just maximizing the number of pairings, the IP method could return a solution with pairings that would be largely unsuited to further assembly. One could adjust the objective function for the IP method to try to replicate this "forward-looking" behavior, but the computational cost deficiency would remain. Another way to circumvent this problem would be to use the IP method to assemble a full column of $s$ stacks at a time, instead of just two. But then the resulting system has $O(N^s)$ constraints, which would quickly exceed the available processing time.

## 5 Results without Scission

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

using this algorithm on the data set provided by the manufacturer. We begin by treating each bin separately, considering it as the initial pool for a single run. (Bin 0 had a very small number of stacks, and hence was excluded from our analysis).

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

**Table 1:** *Comparison of one-step, one-size algorithms for various bins, $q = 400$. To compare performances, we ran each algorithm on the sample data set provided by the manufacturer. This data set consisted of stacks assigned to 9 different bins. For each bin, the difference between the "best" and "worst" algorithm was a single column.*

The assembly protocol was the one-step algorithm outlined in 4.1 with a one-size protocol. Note that this protocol has four permutations of the top and bottom steps. In order to compare their efficacy, we ran each of the four permutations separately for each of the bins; the results are shown in Table 1.

Each method obtained the maximum number of columns for at least one bin, though the number of assembled columns from the different permutations was always within one of the maximum value. Recall that each method selects the first subunit to assemble based upon the measurement ($T$ or $B$) at one end, then selects the second subunit based upon the measurement at the other end. Since the two variables are uncorrelated, which algorithm(s) work best for which bin is essentially random.

Fortunately, since the algorithms run so quickly, it is easy to run all four permutations and simply pick the best solution of the four. In Figure 7 we display the results of such a calculation. 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. For example, bins 7 and 9 have a 30% delay rate, while bin 5 has less than a 2% delay rate. (Several of the bins had relatively few stacks in them; hence the percentage of delayed stacks is somewhat misleading.) Note that except for bins 7 and 9, the algorithm beats the desired standard of 15% delayed stacks.

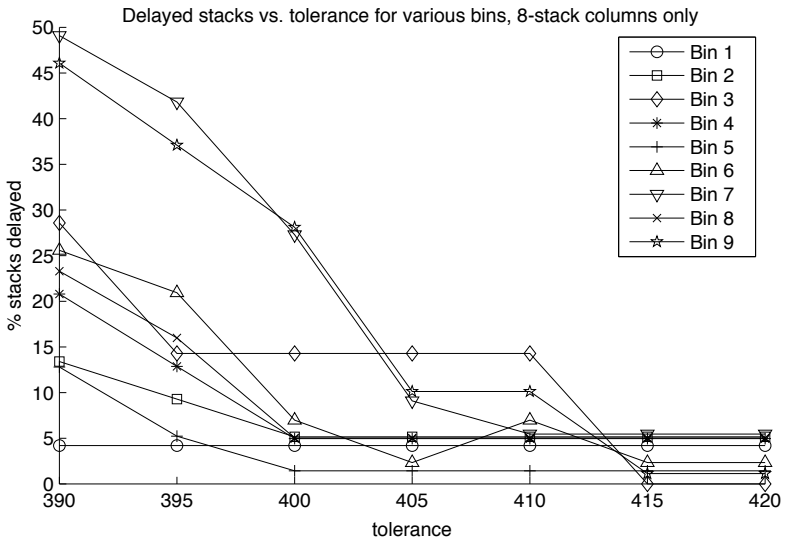


**Figure 7:** *Delayed stacks as a function of tolerance for 8-stack single-bin assembly, one-step algorithm, one-size protocol.*

In general, the delay rate decays as a function of increasing tolerance. 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 by the manufacturer given the distributions in (5). The results in Fig. 7 (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. Thus it is worthwhile for manufacturers to investigate how much the tolerance can be reduced without degrading device performance.

In Figure 8 we present the results of the two-step algorithm described in 4.2 along with the phased one-size approach. Note that in many cases the results compare favorably to the one-step approach.

In all of the previous discussion, we have ignored the problem of forming columns 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. Therefore it is useful to expand the pool of stacks to include adjacent bins, following the categories specified in assembly rule #4.

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

1. Use the one-step algorithm with the one-size protocol to create 4-stack subunits. Do this twice, with initial pools given by bins with adjacent numbers.

2. Combine all the 4-stack subunits from both bins into a single pool and assemble them into 8-stack columns following assembly rule #4(a), (b) (namely, that in a mixed-unit column, the higher-value bin is placed on top).

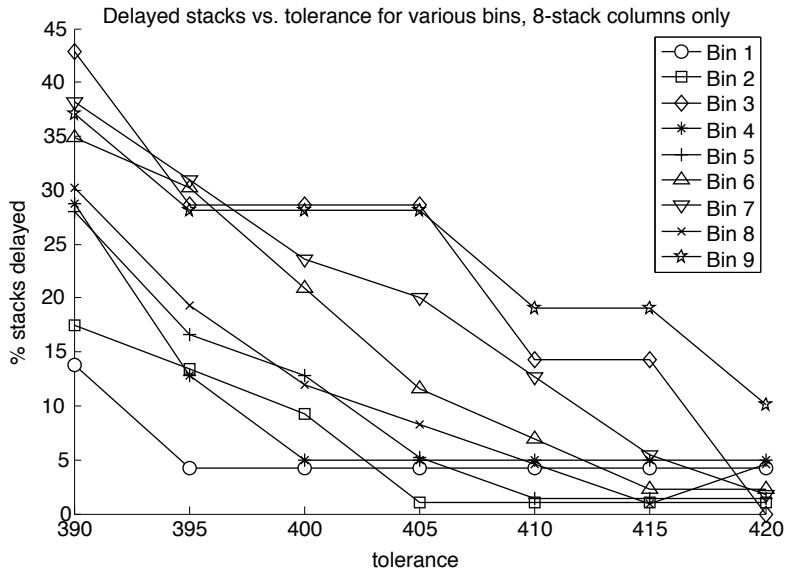


**Figure 8:** *Delayed stacks as a function of tolerance for eight-column stacks, two-step algorithm, one-size protocol, single bins used.*

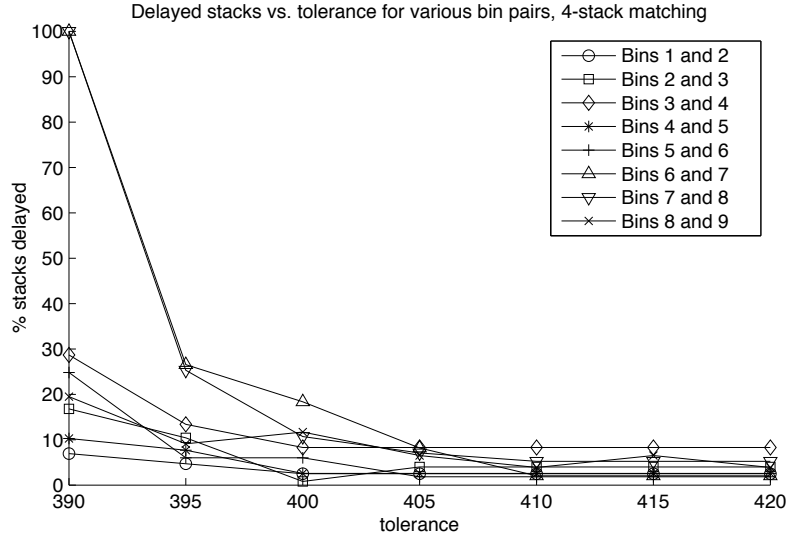


**Figure 9:** *Delayed stacks as a function of tolerance for 4-stack mixed-bin assembly procedure, one-step algorithm, one-size protocol.*

The results are shown in Figure 9. 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.

## 6 Results with Scission

If $s$ is not a power of two, 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 $s + s_e$, where the "e" denotes "extra." Each protocol is defined by its treatment of the following two facets of the scission problem:

1. **Size of subunits**. We explored two possible options for placing $s_e$ extra stacks back into the pool:

   (a) The subunit of length $s_e$ is retained as a single subunit. (Note that $s_e < s$, 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:

   $$\{B_1, T_{s_e}\} \text{ (if removed from bottom) or } \{B_{s+1}, T_{s+s_e}\} \text{ (if removed from top).}$$

   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).) When we wish to remove $s_e$ separate stacks, there is a candidate stack for removal at the top and bottom of the superunit. For each, compute the following quantity:

   $$d_j = \sqrt{\left[\frac{B_j - \bar{B}}{\sigma(B)}\right]^2 + \left[\frac{T_j - \bar{T}}{\sigma(T)}\right]^2}, \tag{8}$$

   where $\bar{B}$ and $\sigma(B)$ are the mean and standard deviation of $B$, respectively, and similarly for $T$. Thus $d$ 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 $d_{s+s_e}$ (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 first present results obtained by using scission with the one-step algorithm when $s$ is not a power of two. In order to compute the case $s = 10$, the following approach was used. Though not particularly elegant, it was easy to implement with the existing one-step code.

1. Use the normal one-step algorithm with one-size protocol to form columns of size 16. The unused subunits from the final step will be eight-stack columns; these will be saved separately as factories often produce 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 scission 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 scission 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 protocol 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.

Moreover, there is an inherent lower bound on the delayed stacks in this method. If the first step of the method works at maximal efficiency, 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 is shown in Figure 10.

When implementing the all-size protocol, scission is required. We compare 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.

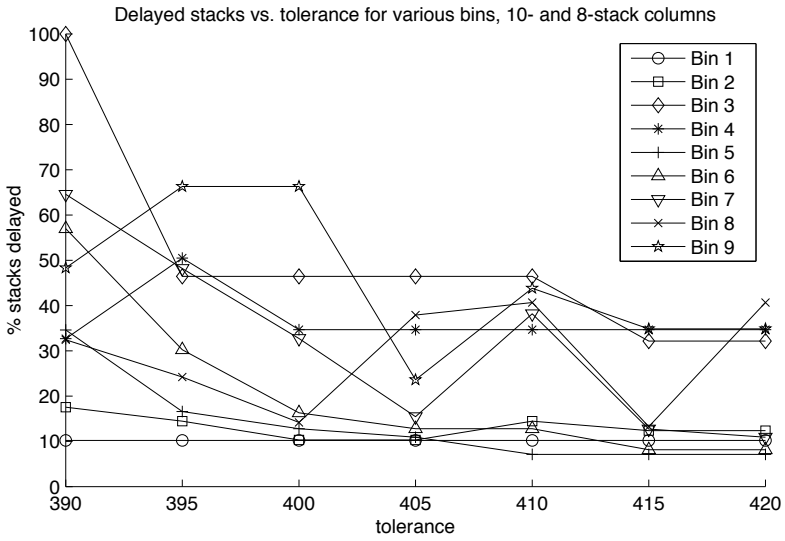


**Figure 10:** *Delayed stacks as a function of tolerance for 16-stack to 10-stack one-size algorithm, one-step algorithm, one-size protocol.*

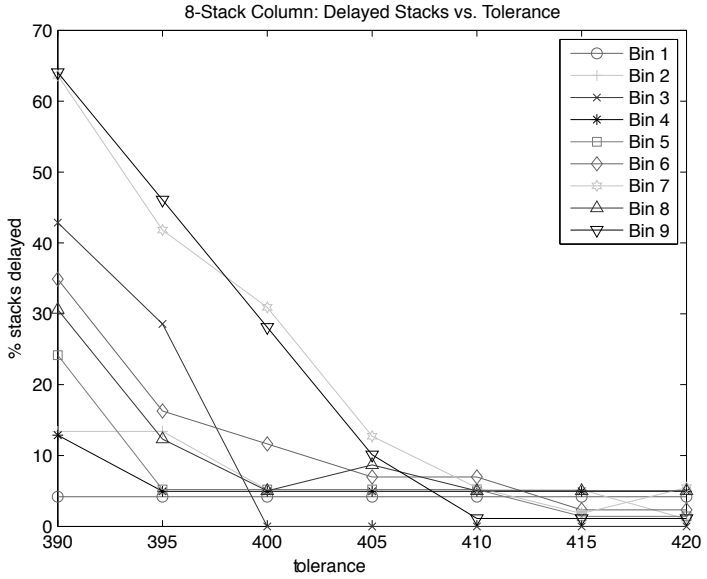


**Figure 11:** *Delayed stacks as a function of tolerance for 8-stack one-step algorithm, all-size protocol.*

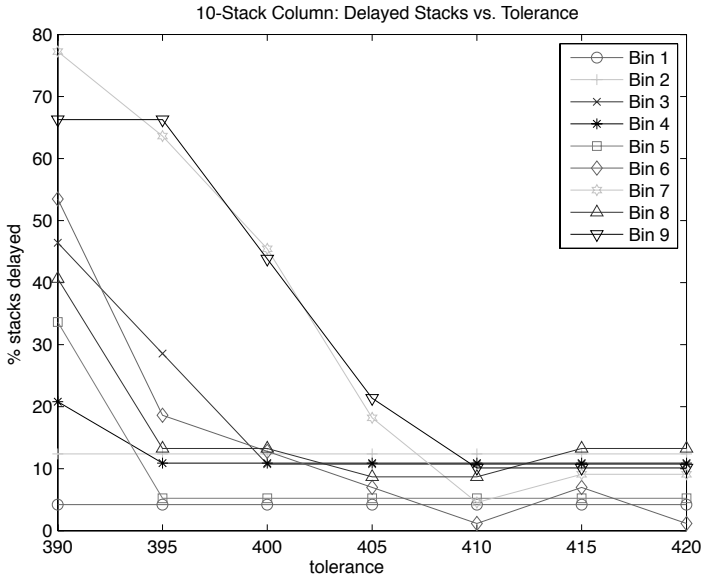


**Figure 12:** *Delayed stacks as a function of tolerance for 10-stack one-step algorithm, all-size protocol.*

Running a 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. 11. Note that the results compare favorably with those presented in Figures 7 and 8 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 Figure 12. Note that the results are clearly superior to those shown in Figure 10. The algorithm can also handle mixed-bin assemblies, but this is beyond the scope of this manuscript.

## 7 Genetic Algorithm

Once several simulations of the assembly process are complete, one can use postprocessing measures in order to improve the results further. In particular, we present results from a genetic algorithm. 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 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* $p_{\mathrm{r}}$ 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 $p_{\mathrm{r}}$. Thus it is likely that there will be multiple copies of certain arrangements, and it is likely that those arrangements were most efficient.

3. **Mutation step**. With some probability $p_\mu$ (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, $p_\mu$ 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 $p_{\mathrm{c}}$ (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 *sub-arrangements* $U_{i,j}$ 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 exchange $U_{1,j}$ with $U_{2,j}$ (the *crossover*) with probability $1/2$.

   With the reshuffling of sub-arrangements, 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 sub-arrangements are retained, though they may be exchanged between arrangements. Thus the range of useful $p_{\mathrm{c}}$ is larger.

6. Now a new generation has been created, so repeat the process.

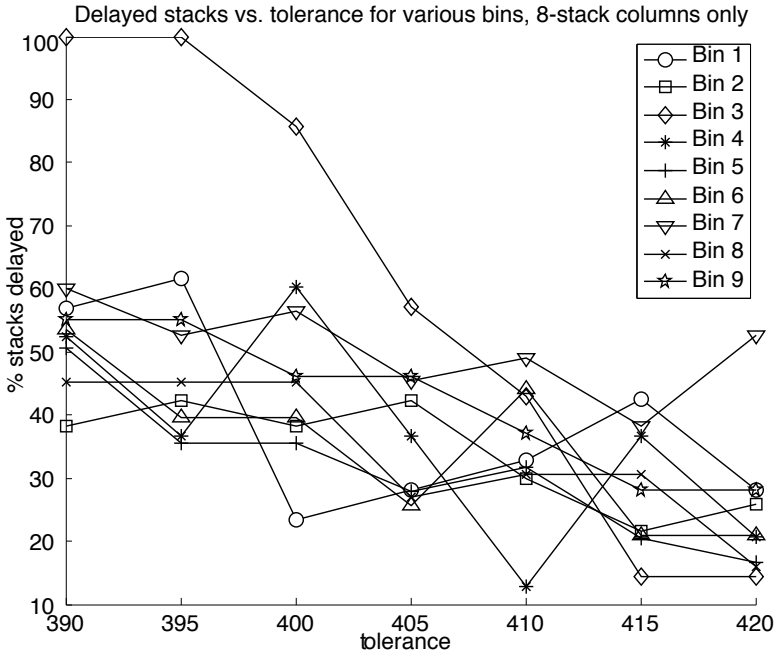


**Figure 13:** *Delayed stacks as a function of tolerance for eight-column stacks, random assembly, genetic algorithm, ten generations. Here* $M = 30$, $p_\mu = 0.01$, *and* $p_c = 0.05$.

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 Figure 13. It shows the delayed-stack percentage of the best solution after ten generations for $M = 30$. In this case, the initial 30 arrangements were generated by a random assembly algorithm, so it is unlikely that near-optimal solutions appeared in generation 0. Hence it is no surprise that the results in Figure 13 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. Therefore, if the arrangements from the heuristic algorithms were used as the first generation for the genetic algorithm, the results would be better than those in Figure 13.

The algorithm is also highly sensitive to the parameters used: $M$, $p_\mu$, and $p_c$. The success of the algorithm depends on those parameters, and the best performance is problem-dependent. (More details about genetic algorithms can be found in [2].) Hence some industrial experimentation would be required to implement this efficiently for the fuel-cell assembly problem.

Nevertheless, the genetic algorithm has 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 probabilistic selection mechanism that focuses the search on regions containing arrangements

with few delayed components.

# 8 Conclusions and Further Research

## 8.1 Conclusions

Given the simplistic nature of the heuristic algorithms presented above, it is perhaps surprising that they work so well. Most of the algorithms were greedy; that is, they built columns iteratively by trying to make a locally optimal choice (chosen differently for the different strategies) at each step. Despite their simplistic nature, all of the assembly methods usually came within a column or two of one another and of the best solution, which has the most possible assembled columns.

We examined a one-size assembly protocol with both a one-step and two-step assembly algorithm. The results were comparable, though the one-step algorithm is easier to implement. Though the one-size assembly protocol is easy to both implement and understand, the all-size protocol usually worked better. This is understandable, 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 all-size protocol is also clearly superior when $s$ is not a power of 2.

Given the time and computational constraints needed to ensure a timely scheduling of the assembly process, an IP formulation was found to be undesirable for two reasons:

1. The number of computations needed depends strongly on the number of components, and even more strongly on the number of subunits simultaneously assembled.

2. If one reduces the number of subunits simultaneously assembled to reduce computation time, the feasible solution returned by the IP algorithm may be more difficult to assemble at the next step than those produced by the greedy algorithms.

Once several simulations have been run, their results may be improved by using 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 better 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.

Manufacturers also want to know how they might modify the assembly rules currently in use to increase efficiency. Our results clearly show that by relaxing the tolerance $q$ given in (2) only slightly, one is able to substantially reduce the number of delayed stacks.

## 8.2 Further Research

Though beyond the scope of this manuscript, for completeness we discuss other facets of the problem which could be analyzed to yield additional assembly efficiencies. First, for quality-control purposes,

manufacturers attempt to avoid mixing components from different vendors. 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 performance of the greedy algorithms generally seemed to improve with larger data sets).

There are other ways to change the initial pool of subunits to assemble. One idea is to work with ranges of the material parameter $A$, rather than the arbitrarily prescribed bins in (1). Though bin limits are arbitrary, bin width is not, and hence bin width prescribes a natural tolerance on $A$. Thus we could replace rule #4 with

$$\max_j A_j - \min_j A_j \leq 0.01 \tag{9}$$

for any column. (Note that the extrema are taken over positions.) The rule given by (9) 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 (9). 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. Therefore, 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.

The smallest possible pool would be $N = cs$, which corresponds to a single box, after which we could add additional stacks as needed if we were unable to assemble a full box. Alternatively, we could take integer multiples of $cs$.

We may also consider 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.

When assembling stacks using the single-step algorithm, there is no reason that the choice of top-step, bottom-step, or sequenced method need be made *a priori*. In particular, at each iteration, one could compare the values of $d$ [as given by (8)] of the candidate stacks produced by the top step and bottom step methods, and choose the method which produces the smaller value.

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 into its component

stacks to see if these smaller stacks can be reassembled into columns.

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. 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.

# 9 Appendix

## Nomenclature

*Variables and Parameters*

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

$A$: parameter related to IC (1).

$B$: parameter related to bottom of stack.

$b$: index related to $B$ value (7).

$c$: number of columns in a box.

$d$: distance metric, variously defined.

$F$: feasibility matrix for IP problem.

$g$: penalty function for IP problem (13).

$i$: integer used to index subunits.

$j$: integer used to index position (2).

$k$: integer used to index subunits (3).

$L(k)$: length of subunit $k$.

$l$: integer used to index subunits.

$M$: integer, variously defined.

$\mathcal{N}$: normal distribution (5).

$N$: 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 parameter value given to anomalous stacks (3).

$q$: tolerance for parameter sum (2).

$r$: number of repeating units in a stack.

$S$: set of subunits in greedy algorithm.

$s$: number of stacks in a column.

$T$: parameter related to top of stack.

$t$: index related to $T$ value (6).

$U$: arrangement in the genetic algorithm.

$\sigma(\cdot)$: standard deviation of $\cdot$ (8).

*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.

$\mu$: as a subscript on $p$, used to indicate the mutation step.

$\bar{}$: used to indicate the mean (8).

$-$: as a superscript, used to indicate minimum (6).

$+$: as a superscript, used to indicate maximum (7).

$*$: as a superscript, used to indicate a crossover arrangement.

## Integer Programming Formulation

To express our problem in the linear programming context, we 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 the number of pairings

$$H = \sum_{i,k} x_{ik} \tag{10}$$

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

$$x_{ik} \leq f_{ik} \text{ for all } i, k. \tag{11}$$

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

$$\sum_{k} x_{ik} + x_{ki} \leq 1 \text{ for all } i. \tag{12}$$

Here the first sum counts bindings where $i$ is on the bottom, while the second sum counts bindings where $i$ is on the top. (For more details on such integer programming problems, see [1] and [5].)

Note that the number of constraints becomes large very quickly. Equation (11) provides $N^2$ conditions, while (12) provides $2N$ conditions. This greatly increases computational time. For instance, using the Matlab `bintprog` optimization function for a system with $N = 167$ stacks took about 5 minutes on a standard laptop to return a set of pairs—the first step of the assembly. In contrast, the 28 combinations outlined in (6) took just 45 seconds to return a set of *columns*.

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 stacks used in the pairings would be different. It can be 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.

This sensitivity to the data reflects the fact that the function $H$ to be optimized is simply the number of pairings; it does not attempt to produce pairings which would be easy to assemble at subsequent stages. In order to do so, one could introduce an objective function of the form

$$H = \sum_{i,k} g(B, T; q) x_{ik}, \tag{13}$$

where $g$ would penalize end values away from the mean, as in $d_j$ in (8). The choice of $g$ is subtle. It must penalize difficult-to-match pairs while ensuring that the maximum number of pairings is still made. The design of such a function is beyond the scope of this manuscript.

In order to work around the issue of designing $g$, one could envision using this approach to join more than two subunits together at once. For instance, suppose we wanted to set up the integer programming system to assemble three subunits at once. Then we would define

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

and the goal would be to maximize

$$H = \sum_{i,k,l} x_{ikl}. \tag{14}$$

The binding constraint then becomes

$$x_{ikl} \leq f_{ik} f_{kl} \text{ for all } i, k, l, \tag{15}$$

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

$$\sum_{k,l} x_{ikl} + x_{kil} + x_{kli} \leq 1 \text{ for all } i. \tag{16}$$

These rules provide $O(N^3)$ constraints. In general, to combine an entire column at once would take $O(N^s)$ constraints.

## Acknowledgments

This work was supported in part by the National Science Foundation, grant DMS-1153940. The authors thank the reviewer for providing useful insights that greatly improved the paper.

## References

[1] J. Franklin, *Methods of Mathematical Economics: Linear and Nonlinear Programming, Fixed-Point Theorems*, Society for Industrial and Applied Mathematics, Philadelphia, 2002. 46

[2] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, New York, 1999. 41

[3] R. O'Hayre, S.-W. Cha, W. Colella, and F. B. Prinz, *Fuel Cell Fundamentals*, Wiley, Hoboken, 2006. 23

[4] C. S. Spiegel, *Designing & Building Fuel Cells*, McGraw Hill, New York, 2007. 23

[5] G. Strang, *Linear Algebra and Its Applications*, Harcourt Brace Jovanovich, New York, 1988. 46