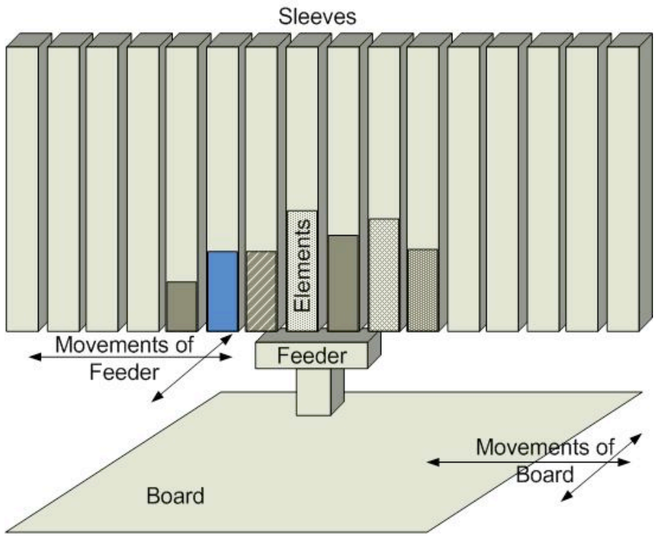
## The Problem

The entire course was centered around a single problem dealing with the manufacturing of printed circuit boards. Printed circuit boards are made up of many different types of components, including transistors, diodes, and capacitors, which are arranged in different numbers in different places on a substrate, or blank board, to form the final printed circuit board. The components are stored in sleeves in a pipe-organ setup above the area where the substrates are held, with a retrieval arm that moves between the sleeves to pick the correct component. Each of these different elements needs to be placed individually on the substrate by a pick-and-place machine. This process involves two steps: first, the correct element must be retrieved from the correct sleeve (corresponding to the type of component needed); second, the substrate needs to be moved below the insertion point so that the element is placed in the correct location.



(Figure 1)

Throughout this class we made the assumption that the time required to move the substrate beneath the insertion point was so small in comparison to the time needed to retrieve the element from the correct sleeve that we were able to ignore it. Thus, we focused our attention on how to minimize the time required to retrieve all the components required for a board. We assumed that the time required to retrieve an element from a given sleeve, the sleeve cost, was monotonically increasing in distance away from the central insertion point, and symmetric on both sides of the insertion point. It can be easily shown that under these assumptions the optimal configuration for any given board assigns the most commonly used component to the least cost sleeve, the second most common component to the second least cost sleeve, and so on an so forth. Thus we can rank the components in terms of their relative frequency, and use this ranking to determine the optimal assignment of components to sleeves. Exactly the same process can be applied to multiple boards—add the frequencies of each component across all boards, rank the aggregate frequencies, and assign components to sleeves based on that ranking.

This is a rather simplified version of the problem, however, because the setups can be torn down and reassigned, for a time cost of σ. We can immediately see that if σ is very large in comparison to the overall production time, then it will be in our best interest to put all the boards together in one setup and manufacture them according to the optimal setup for all boards. If σ is very small, or zero, then we will tear down the setup between every board and manufacture each board according to its own optimal setup. From here on out, we will denote an individual board's optimal setup cost by $f_b^*(b)$ where b is the number of the particular board. The difficulty arises when σ is neither 0 nor exceedingly large, but somewhere in the middle. Now the problem becomes not how to order the components in the sleeves, since the optimal configuration is uniquely determined by the boards being produced together, but exactly which boards to group together into a cluster to be produced under a single setup. Since there are $n$ ways to group $n$ boards into clusters, or sets, of 1 or $n-1$, $\binom{n}{2}$ ways to group those same boards into clusters of 2 or $n-2$, etc., the number of possible ways to group these boards is enormously huge and grows combinatorially. We were given an example with 24 boards, each with 16 components, and we found that the number of possible clusters of size 3 with this data was more than one million. The more you think about the complexity of the problem, the more your head starts to spin—how in the world were we ever going to solve it? This problem is also exceptionally frustrating, because theoretically we know how to solve it – if we had unlimited computational time we could run through all of the possible sets of clusters that encompass all the boards, rank the components by their aggregate frequency in each cluster, assign them to sleeves and compute the cost.

Unfortunately, given the enormously huge number of possible solutions, simply enumerating all of them and comparing their costs would seem to be a foolish approach.

## First Steps

Our first step to try and get a handle on the problem was to try and come up with a good heuristic solution. In this assignment, as in many of the others that involved implementation, the most difficult and time consuming piece was arriving at a suitable data structure. I could not figure out a way to store the clusters as vectors of the boards included, since the dimensions would change from cluster to cluster, so I settled on a system of storing the clusters as a 24-element vector with the first component corresponding to the number of the cluster that contained the first board. This could be arbitrarily set to one, but the random-generation code that I wrote was easier if it could just assign random numbers between 1 and *t* where *t* was the number of clusters into which you wanted to partition the boards. Looking back, much of the implementation would have been easier had I used the system of 24-element vectors of zeros and ones, where a one signifies that that board is in the cluster. But, using my rather clumsy data structures, I was still able to create a randomized loop which would generate 500,000 instances for clusters of size two to twenty-three, and then would save both the average cost across all instances, and the lowest cost along with the vector that generated that lowest cost.

The best answer we were able to generate from this method was 906,231, with an optimality gap of 2.22%. This optimality gap is misleading, however, because we were comparing it with the only lower bound we had—manufacturing all boards alone according to their optimal setup with no tear-down cost—which is clearly an infeasible solution. So in a little over two hours, through our random generation, we were able to get a solution barely 2% away from an impossibly good solution. In reality, our answer was only 0.23% off the actual optimal solution of 904,145. In nearly all industrial applications being 0.23% away from the optimal solution would be more than enough. However, though this worked fairly quickly for the particular data set we were given, we have absolutely no guarantee that it would work for other data sets, or larger data sets. On the other hand, we have seen that having a good feasible solution to start from can result in substantial decreases in the time required to solve the problem, so using a random-generation algorithm such as this might prove useful in obtaining good initial solutions. We also briefly discussed using genetic algorithms in combination with this randomized procedure to generate even better solutions. Unfortunately I do not know enough about genetic algorithms to actually be able to implement them in any consistent fashion, but from the little I do know, my guess is that we could have improved our final solution by breeding several near-optimal solutions together and looking at their offspring over several generations. Additionally, we talked several times about Dushant's local neighborhood search, and the potential it has to help generate good initial feasible solutions. I know even less about neighborhood searches than I do about genetic algorithms, but it seems like it might be an interesting area to look into, especially since outside of the realm of academia, many people are more interested in "good" solutions than in provably optimal solutions. However, being firmly ensconced in the realm of academia, a heuristic solution is not enough to satisfy our unceasing hunger for provable optimality. So we endeavored on.

## The Initial Model

In order to find a provably optimal solution we must first have a formulation. The first formulation we looked at was George Polak's model (the Polak model) from his published paper.

$$\text{min. } \sigma s + \sum_{k\in K}\sum_{i,j\in N} c_j^k x_{ij}^k \qquad \text{...(1)}$$

$$s.t. \quad \sum_{i\in N} x_{ij}^k = 1 \qquad \forall j\in N, k\in K \qquad \text{...(2)}$$

$$\sum_{j\in N} x_{ij}^k = 1 \qquad \forall i\in N, k\in K \qquad \text{...(3)}$$

$$w^{km} \geq x_{ij}^k - x_{ij}^m \quad \forall i,j\in N; k,m\in K; m>k \text{ ...(4)}$$

$$s^k \leq \begin{cases} \sum_{m>k} 1 - w^{km} & for\ k<|K| \\ 0 & for\ k=|K| \end{cases} \qquad \text{...(5)}$$

$$s = K - \sum_{k\in K} s^k \qquad \text{...(6)}$$

$$x_{ij}^k \in \{0,1\} \quad \forall i,j\in N, k\in K \qquad \text{...(7)}$$

$$w^{km} \in \{0,1\} \quad \forall k,m\in K$$

$$s^k \in \{0,1\} \quad \forall k\in K, \qquad s\in Z^+$$

Here $x_{ij}^k = 1$ if component *i* is in sleeve *j* for board *k*, and 0 otherwise; so the first two constraints are simply the assignment problem of components to sleeves. $c_{ij}^k$ is the sleeve cost *j* multiplied by the demand for board *k* times the number of component *i* in board *k*, and *s* is the total number of setups, so the objective minimizes the sum of setup costs and manufacturing costs for all clusters. The model starts to get complicated with the introduction of $w^{km}$: if $x_{ij}^k = x_{ij}^m$ then $w^{km}$ can be either zero or one, but since we're trying to minimize *s* which ultimately depends on $w^{km}$, it will always be zero in an optimal solution. If, on the other hand, $x_{ij}^k \neq x_{ij}^m$, then $w^{km}$ is forced to be one. $s^k$ is defined as the sum of (1-$w^{km}$) for all $m>k$. Thus, $s^k$, in effect, represents the number of boards that share a setup with board *k*. The sum over all *k* of $s^k$ is then subtracted from the maximum number of setups, *K*, where each board is manufactured by itself. This means that each board that shares a setup is subtracted out from the maximum number of setups, leaving the actual number of setups used. It is a fairly ingenious device, but a difficult one to understand at first. We wondered if we would be able to improve on the model, to come up with something that was a little more intuitive.

**New Models**

Since the original model was so difficult to understand, we were given the task of coming up with new models, in the hopes that we could come up with something a little easier to understand. The most intuitive way to construct a model would be to have $x_{ij} = 1$ if boards *i* and *j* are manufactured together. Unfortunately, there is no easy way to determine which boards are in a cluster, nor any way that we could come up with to sort them, given this definition of *x*. Oddly enough, this idea of defining whether two boards are paired together or not turns out to be a very effective branching strategy later on for the master problem, though in a different form. Setting that particular idea aside, Shital and I came up with three models, none of which held any great hope of yielding a solution. Our first model was a slight adjustment to the Polak model, using all the same variables.

*Model 1*

$$\min \quad \sigma s + \sum_{i\in N}\sum_{j\in N}\sum_{k\in K} c_{ij}^k x_{ij}^k$$

$$\sum_{i\in N} x_{ij}^k = 1 \qquad \forall\, j\in N, k\in K$$

$$\sum_{j\in N} x_{ij}^k = 1 \qquad \forall\, i\in N, k\in K$$

$$w_{km} = -\left|x_{ij}^k - x_{ij}^m\right| + 1 \qquad \forall\, k\in K, m\in K, k\neq m$$

$$s^k \leq \sum_{m>k} w_{km} \qquad \forall\, k\in K$$

$$S = K - \sum_{k\in K} s^k$$

$$x_{ij}^k \in \{0,1\},\ w_{km}\in\{0,1\},\ s^k\in\{0,1\}$$

This model presents a slightly more intuitive definition of $w_{km}$ where $w_{km}=1$ if boards *k* and *m* are in the same cluster. Unfortunately, in order to achieve this, we had to introduce a non-linearity into the model in the form of absolute value. So while it may be slightly more intuitive, the non-linearity excludes it from being truly useful in solving the problem. But thinking about this model led us to our second model which eliminates the issue of determining how many setups there are by assuming that we know there are *S* setups. Thus, this second model would need to be solved once for *S*=1,…,*N*, where *N* is the number of boards.

*Model 2*

$$\min \quad \sigma S + \sum_{i\in N}\sum_{j\in N}\sum_{k\in K} c_{ij}^k x_{ij}^k$$

$$\sum_{i\in N} x_{ij}^k = 1 \qquad \forall\, j\in N,\ k\in K$$

$$\sum_{j\in N} x_{ij}^k = 1 \qquad \forall\, i\in N,\ k\in K$$

$$\sum_{r\in S}\left(1 - p_r^k\right) = 1 \qquad \forall\, k\in K$$

$$p_r^k + p_r^m \geq x_{ij}^k - x_{ij}^m \qquad \forall\, r\in S;\ k,m\in K;\ k\neq m$$

$$x_{ij}^k \in \{0,1\},\ p_r^k\in\{0,1\}$$

Here $p_r^k = 0$ if cluster *r* contains board *k*, and 0 otherwise, returning us to the counterintuitive '0 means yes' definitions from the earlier Polak model. The first two constraints are the same as in the Polak model, and the third simply says that each board must be contained in exactly one cluster. The confusion arises from the fourth constraint; however, if we look more closely we can see that the constraint is only meaningful in the case when $p_r^k = p_r^m = 0$, i.e. boards *k* and *m* are in the same cluster, thus their setups must be the same. This is exactly what is implied by the second part of the constraint, since then it must be true that $x_{ij}^k = x_{ij}^m$, and whether they both are zero or one is immaterial. Additionally, when $p_r^k \neq p_r^m$ the constraint drops out and becomes meaningless. Thus, this fourth constraint guarantees that all boards in the same cluster have the same setup. This model seems much more elegant and intuitive than the original model. Unfortunately, it needs to be solved *N* times (where *N*=number of boards). As we were running the random-generation code, we noticed that nearly all of the best solutions had between six and nine clusters. If this was a structure that was common across all data sets, a formulation such as this one would allow us to exploit that and only look at the solutions with the likely number of clusters. However, in order to be able to prove that a solution obtained in this manner is optimal, we must be able to show that the function of optimal solution value vs. number of clusters in the solution is convex, which we will address in the next section. The final model we tried took an entirely different view of the problem, setting a new variable $x_{bs} = 1$ if board *b* is manufactured according to setup *s*, and 0 otherwise.

*Model 3*

$$\min \quad x_{bs} y_{bc} c_{cs} + \sigma T$$

$$w_s \geq 1 - \sum_{b \in B} x_{bs} \qquad \forall\, s \in S$$

$$\sum_{s \in S} x_{bs} = 1 \qquad \forall\, b \in B$$

$$T = S - \sum_{s \in S} w_s$$

$$x_{bs} \in \{0,1\},\ w_s \in \{0,1\}$$

Here, $y_{bc}$ is the number of component *c* in all the boards of type *b*, $c_{cs}$ is the cost of component *c* in setup *s*, and *S* is the total number of setups (*N*!). Furthermore, $w_s = 1$ unless there is at least one board manufactured according to setup *s*. This is perhaps the most easily understood model we looked at. Regrettably it has *B(N!)* variables, and nearly as many constraints. In a model with 24 boards and 16 components, this model will have $502 \times 10^{12}$ variables and roughly the same number of constraints. This is simply too many variables and constraints to think about realistically solving.

Throughout the process of trying to come up with alternate formulations for this problem we realized just how difficult it can be to effectively model a problem that can be described in only a few sentences. Later, as we looked at modeling the subproblem, this was really hit home—translating problems into math programs can be exceedingly difficult, even when the problem itself seems relatively easy to describe.

## **Convexity of the Solution Set**

If we knew that the optimal value was convex in the number of clusters in the solution we could take formulations such as *Model 2* above, and solve them for increasing numbers of clusters until we saw an increase in function value. If the function is convex, once you see an increase the function value will continue to increase, implying that our previous solution is, in fact, optimal. This has the potential to save a great deal of computation time if the optimal number of clusters is small relative to the number of boards. If the optimal number of clusters is close to the number of boards, then even if the function is convex, it will not save us a great deal of time since we will have to solve the problem for all, or nearly all values of $S$. Working with Shankara, we tried to find a counterexample using a set of only three boards with three components. Our flaw was assuming symmetric sleeve costs; we tried all possible combinations of boards and found that all of them (under the assumption of symmetric sleeve costs) were convex. Others in the class rejected the assumption of symmetric sleeve costs and thought they had found a counterexample, though it was later proved to be incorrect. Further, their counterexample was an extremely pathological case, with strongly asymmetrical sleeve costs. All the evidence we could find supported the conclusion that with symmetric sleeve costs, the function would indeed be convex; but we were unable to provide a proof, so it is all merely conjecture. Yet even if we could prove convexity, it would mean that we would need to solve an impossibly difficult problem slightly fewer times, but we still do not know how to solve the problem even for a given number of clusters.

**Integrality of Solutions**

Another dimension of the solution involves the integrality of the solution. Clearly we cannot have fractional amounts of clusters in an optimal solution; but solving the linear programming relaxation of the problem is vastly easier than solving the integer program. The question then becomes: If we solve Polak's model with the *x* variables relaxed, will the solution still be integer? If the constraint matrix is totally unimodular (TU) then yes, the solution to the LP relaxation will be integer. If the constraint matrix is not TU, then the solution may or may not be integer. So we set out to prove that the constraint matrix of the Polak model was TU. Unfortunately, our proof turned out to be flawed and we were unable to correct for the flaw, though it does seem that in most instances the constraint matrix will be totally unimodular. At the end of the course, looking at the solutions using the rank-cluster-price algorithm, most of them did turn out to be integer, especially for small problem instances, implying that we might well be correct that the formulation is indeed TU. Regardless, the solutions do turn out to be integral in many instances.

The integrality of the solution is related to the strength of the LP relaxation. If the LP relaxation is close to the convex hull of the integer program, then the solution will often be integer. Another way of conveying the idea of total unimodularity, is that a constraint matrix that is TU describes exactly the convex hull of the integer feasible region, and so all the extreme points will be integer. In fact, we do not particularly care if *all* the extreme points are integer, only if the extreme points near the optimal solution are integer. But we were unable to prove that even just the extreme points near the optimal solution were integer, though it might be a more efficient way to approach the proof. Regardless, the better your LP relaxation is, the fewer nodes there will be in your branch and bound tree once you start trying to solve the integer program. As we saw later, this can have a profound impact on the solvability of a problem.

**Dantzig-Wolfe Decomposition**

Having nearly exhausted the store of knowledge in the class without additional information, we turned our attention to Dantzig-Wolfe decomposition, which motivates the technique known as column generation. The idea is that we can separate the problem into a set of subproblems which are then linked together in a "master problem." Typically these subproblems have some special structure, such as an assignment problem, that allows them to be solved quickly and easily. This technique is especially useful if you have a set of constraints that affects only a small group of variables, thus those variables and constraints can be separated out and put into subproblems, with "linking constraints" in the master problem which link all the subproblems together. These subproblems then generate a set of extreme points of the solution. And we know that the optimal solution can be expressed as a convex combination of the extreme points of the subproblems, so we can replace the variables in the master problem with these convex combinations of extreme points, and solve. If the subproblems are easily solved, this can prove to be a very powerful mechanism for solving large-scale problems. The key, however, is that the subproblems need to be easy to solve.

## Subproblem Formulation

With the column generation technique, motivated by the Dantzig-Wolfe decomposition, in mind we set out to define a master problem and subproblem to which we could apply these ideas. We fairly quickly found our master problem:

*Master Problem*

$$\min \sum_{k \in K} p_k x_k$$

$$\sum_{k \in K} \delta_{bk} x_k = 1 \quad \forall\, b \in B$$

$$x_k \in \{0,1\}$$

Here, we change notation slightly where $K$ is the set of all clusters being considered. $x_k = 1$ if cluster $k$ is in the optimal solution, and 0 otherwise. $p_k$ is the cost of cluster $k$ including setup cost σ. $\delta_{bk}$ is an index variable which equals one if board $b$ is in cluster $k$. Thus the only constraint in our master problem is that each board must be contained in exactly one cluster, and we want the set of clusters with minimum cost. It seems simple enough, and the idea behind the subproblem is fairly simple. We even know what the subproblem is: to generate the most negative reduced cost column. Then we would take that column from the subproblem, enter it into the master problem, get new duals from the master problem and hand those off to the subproblem and repeat until our subproblem reports back that there are no negative reduced cost columns, showing that our current solution is optimal. Regrettably, this turns out to be much easier said than done.

The reduced cost of a column (cluster) is its true cost minus the dual associated with the boards included in that cluster. The true cost is the manufacturing cost of all the boards included in the cluster plus the setup cost, σ. Thus, we define variable $p_{clb} = 1$ if component *c* is retrieved from location *l* for board *b* in this particular cluster, and 0 otherwise. Additionally, we use $x_b = 1$ if board *b* is included in the cluster, and $y_{cl} = 1$ if component *c* is assigned to location *l*. This results in the first version of the subproblem,

*Subproblem Model 1*

$$\min \quad \sigma + \sum_b \sum_c \sum_l d_{bc} c_l p_{clb} - \sum_b \pi_b x_b$$

$$\sum_l y_{cl} = 1 \qquad \forall c$$

$$\sum_c y_{cl} = 1 \qquad \forall l$$

$$p_{clb} \leq y_{cl} \qquad \forall b, c, l$$

$$\sum_l p_{clb} = x_b \qquad \forall b, c, l$$

$$x_b, y_{cl}, p_{clb} \in \{0,1\}$$

This model minimizes the reduced cost, subject to the following constraints: each component is assigned to exactly one location; each location is assigned exactly one component; if component *c* is not assigned to location *l*, then component *c* cannot be retrieved from location *l* for any board *b*; finally, if $x_b$ is one then each component must be retrieved from a location for board *b*. Together these constraints define a feasible column, and the problem will find the feasible column with the most negative reduced cost. Unfortunately, this problem would not yield an answer in less than 2 hours with either the *x*, or the *y*, or the *x* and *y* variables relaxed. When we relaxed all the variables, we ran into a different problem: the LP relaxation is weak. When you solve the LP relaxation with all variables (*x, y, p*) relaxed, the optimal solution is fractional. The reason the solution turns out to be fractional, while not immediately obvious, is because the model is splitting the components for the boards it wants into different sleeves and only paying for the cheapest one. And while it will solve to an integer solution with some dual values, we only got it to solve with made-up dual values, or the dual values from the near-optimal solution. Unfortunately, after only a few iterations of shuffling between the master and subproblem starting with the near-optimal solution, the duals become such that the subproblem will no longer solve to integrality. In fact, even with all the variables relaxed we could not get it to solve with the duals from solving the master problem with the identity matrix.

All these frustrations drove us to look at an entirely different model,

*Subproblem Model 2*

$$\max \quad \varepsilon$$
$$\sum_l x_{cl} = 1 \qquad \forall\, c$$
$$\sum_c x_{cl} = 1 \qquad \forall\, l$$
$$\sum_l \sum_c \delta^b_{cl} x_{cl} - \pi_b + \theta_b \le M(1 - y_b) \quad \forall\, b$$
$$\theta_b \le My_b \qquad \forall\, b$$
$$\sum_b \theta_b \ge \varepsilon + \sigma$$
$$x_{cl}, y_b \in \{0,1\}, \theta_b \ge 0, \varepsilon \ge 0$$

This model is exceedingly difficult to understand. In this model the $x_{cl}$ variables take the place of the $y_{cl}$ variables in the previous model, the $y_b$'s replace the $x_b$'s, and the $\delta^b_{cl}$ is data simply taking the place of $d_{bc}c_l$ in the previous model. When $M$ is large enough, and $y_b$ is one, implying that the board is in the cluster, the constraint says that $\sum_c \sum_l \delta^b_{cl} x_{cl} - \pi_b + \theta_b \le 0$ or, that $\theta_b$ has to be at most the inverse of board $b$'s contribution to the reduced cost of the cluster. And since we're trying to maximize ε (or – the reduced cost of the cluster) which is constrained to be less than $\sum_b \theta_b - \sigma$ we would like all the $\theta_b$'s to be as large as possible, or equal to the board's contribution to the reduced cost of the cluster. Additionally, if $y_b$ is zero, then the next constraint kicks in and $\theta_b$ must also be equal to zero; if $y_b$ is one, then the constraint drops out.

Unfortunately, as we have seen, introducing a big $M$ into a model almost always results in a terrible LP relaxation. Here, the problem arises because the model has two constraints to determine the value of $\theta_b$: $\theta_b \le M(1 - y_b) - \sum_c \sum_l \delta^b_{cl} x_{cl} + \pi_b$ and $\theta_b \le My_b$.

If *M* is truly large, then we can assume that $\sum_c \sum_l \delta_{cl}^b x_{cl} + \pi_b$ is small in comparison with *M(1-$y_b$)*, so the constraints can be reduced to $\theta_b \le M(1 - y_b)$, $\theta_b \le My_b$. We can assume that $\theta_b \ge 0$ because we will never make boards with a positive contribution to the overall reduced cost. We can easily see that $\theta_b$ will be largest when both the constraints are the same, since otherwise one could be pushed a little bit making the value of $\theta_b$ a little larger. Thus, there is strong incentive to make both constraints equal, which will happen with $y_b$ close to ½, so $y_b$ will never be binary in the LP relaxation. In fact, we showed that we will need to branch on every single value of $y_b$ before we will arrive at an integer solution.

Despite all these problems, we were able to get a single instance of this version of the subproblem to solve with the duals from the identity matrix in just over twenty minutes. But returning to the big picture, this means it takes twenty minutes to solve one instance of the subproblem to optimality. In order to solve the root node of the master problem, we might need to solve the subproblem millions, or tens of millions, of times. And then, we have no guarantee that the solution to the root node of the master problem will be integer, so we might have to branch on the master problem, meaning we would have to solve potentially hundreds or thousands of versions of the master problem before we arrived at a truly integer solution to our overall problem. Given the complexity of the overall problem, a subproblem that takes twenty minutes to solve is simply worthless in the overall context. And since there is really no way to remove the big *M* from the model, we were forced to chuck this rather creative model, returning our focus to the seemingly less-promising *Subproblem Model 1*.

## Subproblem Trials

Looking at *Subproblem Model 1*, we noticed that the $p_{clb}$ variables were simply taking the place of $x_b y_{cl}$, and serve only to eliminate the nonlinearity in the model. So what if we left the nonlinearity in the model? We end up with a new, nonlinear model:

<u>*Nonlinear Model*</u>

$$\min \quad \sum_c \sum_l \sum_b d_{bc} c_l x_b y_{cl} - \sum_b \pi_b x_b$$

$$\sum_l y_{cl} = 1 \qquad \forall c$$

$$\sum_c y_{cl} = 1 \qquad \forall l$$

$$x_b \in \{0,1\},\ y_{cl} \in \{0,1\}$$

This model boils down to an assignment problem, which anyone can understand quickly. Additionally, assignment problems typically solve very quickly. We used the MINOS package to solve this model, and the subproblem solved in mere seconds. Shuffling back and forth between the subproblem (in MINOS) and the master problem (in CPLEX) was regrettably too time consuming to really be able to see if the nonlinear model would yield an answer to the subproblem, however it does appear that it would be a good method to generate good columns quickly. To prove optimality in any of our other models, we need to show that there exist no more negative reduced cost columns. The worry is that this nonlinear model might return a positive reduced cost column when there were still negative reduced cost columns out there. But at the very least, we would only have to solve the long linear model a few times, possibly only once, to verify that the nonlinear model had given us all the available negative reduced cost columns. Unfortunately, before we can actually test this theory, we would need to be able to write a .run file which would allow us to switch between MINOS and CPLEX to solve the subproblem and master problem, respectively, without having to do it by hand. It would seem as though there would have to be a way to accomplish this, although we were unable to find anyone who knew how to tackle this particular issue. With a better implementation, this method would seem to hold great promise for solving this problem quickly. I think this is perhaps one of the most exciting avenues we ignored.

Since we could not find a way to get the nonlinear model to solve automatically, we were forced to return to our original model, *Subproblem Model 1*. We could not get *Model 1* to solve with the duals from the identity matrix, so we tried solving it with the duals from the many iterations we did by hand using the nonlinear model, thinking that these duals might be "better" in some sense. Unfortunately, they did not yield a solution either. Then we had the brilliant idea of using the near-optimal solution from the first week as a starting solution for our master problem—this eight column solution gave us duals that allowed the subproblem to solve lickety-split. This solution raised issues about the validity of the duals from a solution that did not form a basis. In order to form a basis, you need as many variables as you have constraints; here we had only eight variables, one for each column (or cluster) and twenty-four constraints, one for each board. What does this mean for the validity of the duals? After much thought, we decided that even though we do not have a full basis to start with, the duals are nonetheless meaningful, since we end up with exactly eight (the number of columns we started with in the initial solution) non-zero duals. All the other duals, corresponding to the extra variables that are needed to form a basis are simply set to zero. However, this does yield a strongly degenerate solution, where most of the variables in the solution are set to zero. This means that when we took those duals and plugged them into the subproblem the objective value stayed exactly the same over many iterations, because of the degeneracy. The algorithm was simply pivoting in different variables, which were all set to zero, so the objective function did not change. This is the definition of a degenerate pivot. Degeneracy kept coming up again and again throughout this course, and often drastically increased our computation time.

After a few iterations of switching back and forth between the master and subproblem, starting with the near-optimal solution, eventually resulted in duals that the subproblem could no longer solve quickly, at roughly the same point as it reached a full rank basis. We hypothesized that the speed of the subproblem was due to the reduced size of the problem—with only eight non-zero duals, you really only have to consider eight boards, which makes the problem much faster. And indeed when we solved the master problem with the eight near-optimal columns plus the identity matrix, the subproblem took more than a few hours to solve, supporting our hypothesis that it was the zeros in the duals which made the near-optimal instance easier to solve. So how are we ever going to get this problem to solve? We need it to give us an integer solution, and we have already seen that the LP relaxation will always give us $x_b$ values close to ½, so eventually we're going to have to put it into a branch and bound tree. Perhaps looking at the branching strategy will give us some insight, we thought.

## **Branching Strategies**

When we started thinking about branching, little did we know we would be spending weeks and weeks on it, and what a big deal it would turn out to be. Our first step in looking at branching was to observe what happened when we branched on different variables. We initially started branching on the $x_b$ variables, since they seemed to be the ones we ultimately cared the most about—whether or not a board was included in a cluster. Branching on the *x*'s changed the one particular value you set to be integer, but left all the other values fractional. As soon as you set the very last *x* to be integer then all the *y*'s and *p*'s would suddenly become integer, but you need to get all the way to the bottom of the tree before you seen any major changes, which could result in $2^{24}$ (in our instance with 24 boards) branches before you get a solution. $2^{24}$ is a large number, and not something you want to sort through every time you want to solve the subproblem (which might need to be solved millions of times to generate an answer to the root node of the master problem, which may or may not be integer). So we decided branching on the *x*'s was probably a bad strategy, all things considered.

If branching on the *x*'s will not work, then what about the *y*'s? Setting a $y_{cl}$ variable to be one means that component *c* is assigned to sleeve *l*, which automatically sets all other *y*'s in that row and column to be zero, since only one component can be assigned to each sleeve and vice versa. Additionally, you also set all $p_{clb}$ values containing that component or location, but not in conjunction with each other, to zero. If component *c* is assigned to sleeve *l*, *c* cannot be retrieved from any other location for any board; likewise no other component can be retrieved from location *l* for any board. So branching on the *y*'s holds promise. Unfortunately, when we look at setting a *y* variable to zero, almost nothing happens; we see an infinitesimally small decrease in the objective function, nothing like what you see when you set *y* to be one. This is a potentially hazardous branching strategy, since it results in a strongly unbalanced tree. And here again we have the same problem that we need to completely enumerate the *y* values before we see the *x* values become integer. The objective jumps the most when setting *y* to be one, and the least when setting *y* to be zero, so perhaps there is no clear cut winner. If we have to completely enumerate the *y*'s it will only be $2^{16}$ instead of $2^{24}$ branches, but $2^{16}$ is still too many branches for a subproblem. But what about the *p*'s?

Setting $p_{clb}$=1 has an even greater impact than setting $y_{cl}$=1, since $p_{clb}$=1 then sets $y_{cl}$=1 with all the concomitant effects, in addition to setting $x_b$=1, and many other $p_{clb}$'s to zero. Surely this is the largest impact that any one variable can have on the problem. But there are $24*2^{16}$ $p_{clb}$ variables, and again if we look at the other side of the tree, where we set $p_{clb}$=0, there is practically no change in the objective, and no other variables are set to be integer. In the end, there is no clear winner—all the branching strategies have their flaws, and the problem was starting to look completely hopeless. Since we were not getting anywhere with this approach we decided to go back and see if we could figure out why the optimal solution was fractional and if we could fix it in any way.

## **Exploiting Problem Structure**

We had seen earlier that the reason the LP relaxation was fractional was because the model wanted to split the boards between several different columns and then only pay for the cheapest one. We noticed, as we were working through the branching strategies, that when you set $y_{cl}$ to be zero, meaning that you can no longer assign component $c$ to sleeve $l$, it did not change the objective value, because the model simply moved all the components that were in sleeve $l$ and moved them to the symmetric sleeve that had the same cost as sleeve $l$, changing nothing, really. This means that we were branching twice as many times as we really needed to. So how can we eliminate this waste? By creating a model that assigns two components to each sleeve, and only has half as many sleeves; then you can randomly pick which of the two components goes in each of the paired sleeves because they result in equivalent costs. This lead us to the next model.

*Non-symmetric Model*

$$\min \quad \sum_b \left[ \sum_c \sum_l d_{bc} c_l p_{clb} - \pi_b x_b \right] + \sigma$$

$$\sum_l y_{cl} = 1 \qquad \forall\, c$$

$$\sum_c y_{cl} = 2 \qquad \forall\, l$$

$$p_{clb} \le y_{cl} \qquad \forall\, c,\, l,\, b$$

$$\sum_l p_{clb} = x_b \qquad \forall\, b,\, c$$

$$x_b,\, y_{cl},\, p_{clb} \in \{0,1\}$$

This model differs from *Subproblem Model 1* only in that the sum of $y_{cl}$ over all components for each location is two here, instead of one as in the previous model. Everything else remains the same. Since this problem has only half as many sleeves, the number of variables is reduced by just less than half, and results in roughly half as many branches as the original, the assumption was that it would solve faster. We ran a check to make sure that both models gave us the same answer when applied to a smaller, randomly generated data set, and indeed they did, showing that the models, in their integer forms, are equivalent. Hopefully the LP relaxation of the non-symmetric model will be stronger than that of the original. Unfortunately, even with the fewer number of variables, we were still unable to get the non-symmetric model to solve with any of the variables set to be integer using the duals from the identity matrix, implying that while it may be faster, it is still not fast enough. So again, we examined potential branching strategies. Here, instead of looking at what happened near the top of the tree following a breadth-first search, we decided to see what happened near the bottom of the tree and followed a depth-first search. We confirmed that indeed none of the $x$ values became integer until all the $y$ values had been set. More than that, we discovered that there was an order of magnitude jump between the next-to-last branch and the very last branch where all the values suddenly became integer. This is unfortunate because it means that pruning will be nearly impossible—none of the nodes in the branch and bound tree will ever be above the upper bound of the incumbent solution until they are themselves integer—implying that we are still facing the potential of full enumeration to solve the subproblem. We also tried branching on the $p$ variables, but since every time we set a $p$ to be one we were forcing the corresponding $x_b$ and $y_{cl}$ variables to be one as well, we were in effect randomly picking $x$'s, which resulted in a very poor solution. Again we saw the problem of the imbalance in the tree – setting whichever variable you choose to be one has a far greater impact that setting that same variable to zero, so the tree is much larger on the zero side. Eventually we decided that examining the branching strategy might not actually be fruitful, and the course came full circle as we turned our energies back towards heuristics.

## **Heuristics**

Heuristics are generally not considered as elegant as traditional linear programming methods, but we were hoping that we could find a heuristic that would generate negative reduced cost columns quickly. This might be an effective strategy because we do not particularly care about finding the most negative reduced cost column; we simply need a single negative reduced cost column (or many if we want to put in multiple columns at a time). The first thing we noticed was that we can automatically throw out boards with negative or zero duals, since there is no way their contribution to the reduced cost can be negative, since we still need to pay for the manufacturing. We also know that including boards with larger duals will be more likely to yield a negative reduced cost.

The first heuristic we came up with started by sorting the duals in descending order. We automatically included the two boards with the largest duals, since we know any new column will have at least two boards (since we started with the identity matrix where each board is by itself). Then, in decreasing order of the duals, we examine each board. If adding the board to the previous cluster maintains a negative reduced cost column, we add it and move on to the next board. If the board makes the reduced cost greater than zero, then we do not include the board and output the previous cluster. This would seem to be a smart heuristic because it will almost always yield a negative reduced cost column, but it fails to take into account any measure of similarity between the boards. If boards 1 and 2 have very large duals but are entirely dissimilar, it might not be a good idea to put them together into a cluster. Additionally, we discovered that this heuristic generated very similar columns at every iteration, because the duals do not change much when you start with the identity matrix. We saw other situations where the duals changed much more rapidly, so there is a chance that this heuristic would work in different circumstances; however, it is not a good bet when starting from the identity.

Our second heuristic harkened back to the random-generation code of the very beginning. We simply generated random clusters, checked their reduced cost, and saved the smallest. If the random generation failed to produce a negative reduced cost column, then we applied our first heuristic. Though we consistently produced negative reduced cost columns, the objective value was decreasing by only a little more than 1% per iteration, so using this method would take a long time to reach a near-optimal solution.

The other idea motivating us to find good initial solutions was the thought that we could then use them in AMPL as better upper bounds, allowing CPLEX to prune the branch and bound tree faster. What we discovered with both of our heuristics was that AMPL generated an integer solution with a better objective value than that of our heuristics in less than a minute, so we were not really improving the pruning by giving it our heuristic upper bound. Perhaps had we been able to generate solutions with much lower objective values it would have made a difference, but given what we had, we made no impact at all. AMPL is much smarter than we give it credit for, it would appear.

Still fixating on the issue of solving the subproblem with the duals from the identity matrix, we decided to test whether or not it was a special property of the identity matrix, or simply the fact that we had a full rank basis. So several people tried multiple randomly generated sets of 24 clusters, and found that often the clusters did not provide a feasible solution. Adding the columns from the near-optimal solution so that a feasible solution existed, however, resulted in a problem that still took just as long to solve as the one using the duals from the identity matrix. Thus, we can firmly state that the faster speeds observed using the near-optimal solution stem not from the near-optimality of the solution, but the fact that the columns do not form a full-rank basis. This exercise did not really provide any profound insight, but at least confirmed that our suspicions were correct.

So now we've decided we cannot come up with a way to generate good upper bounds (or at least upper bounds that are better than those that AMPL arrives at within minutes), we cannot find any inspiration from branching strategies, and we are sure that this is our most promising model. It is at this point that we were all grateful that this was not *our* dissertation research. So what do you do?

## **On the Road to a Solution**

Thankfully, we had our intrepid professor to help us along, and refocus our attention on pruning. Previously, we had agreed that any board with a negative or zero dual would not be included in an optimal solution, but perhaps there was a tighter bound we could use. If we define $f_C^*(b)$ to be the cost of manufacturing board *b* according to the optimal setup for cluster *C*, we know that $f_b^*(b) \le f_C^*(b)$ since manufacturing any board by itself will be at least as good as manufacturing it with other boards. This means that $f_C^*(b) - \pi_b \ge f_b^*(b) - \pi_b \quad \forall C$. Thus, $f_b^*(b) - \pi_b$ provides an lower, or best, bound on the contribution of board *b* to the reduced cost. So not only can we ignore boards where $\pi_b \le 0$, we can also ignore boards where $\pi_b \le f_b^*(b)$ since the actual manufacturing cost will be at least $f_b^*(b)$, and if the dual is not large enough to outweigh the lower bound on the manufacturing cost there is no way the actual contribution to the reduced cost will be negative. This gives us a potentially smaller set of variables to start with, but we still have no effective way to prune the tree.

Perhaps the most intuitive pruning technique is to look at the tree and see if adding a board to a particular cluster increases the reduced cost. If so, then you can prune that branch of the tree. But this is only a valid method of pruning if you can prove that there is no other combination of boards including the original cluster, the additional board and some subset of the remaining boards which will have a lower reduced cost than that of the original cluster. In other words, can we prove that if $\sum_{b \in C} (f_C^*(b) - \pi_b) < \sum_{b \in C} (f_{C \cup i}^*(b) - \pi_b) + f_{C \cup i}^*(i) - \pi_i$ for some additional board *i*, then it will always be true that $\sum_{b \in C} (f_C^*(b) - \pi_b) < \sum_{b \in C} (f_{C \cup g}^*(b) - \pi_b) + \sum_{b \in g} (f_{C \cup g}^*(b) - \pi_b) \quad \forall g, i \in g$ where *g* is a subset of the remaining boards containing board *i*? We approached this by looking for a counterexample. We used boards 1 and 2 as our initial cluster, board 3 as our '*i*' board that we add, and then boards 3, 4 and 5 as our *g* cluster. In this case, the first equation becomes $f_{1,2}^*(1) + f_{1,2}^*(2) - \pi_1 - \pi_2 < f_{1,2,3}^*(1) + f_{1,2,3}^*(2) + f_{1,2,3}^*(3) - \pi_1 - \pi_2 - \pi_3$ which reduces to $f_{1,2}^*(1) + f_{1,2}^*(2) < f_{1,2,3}^*(1) + f_{1,2,3}^*(2) + f_{1,2,3}^*(3) - \pi_3$. But we also need the second equation to be false, which means

$$f_{1,2}^*(1) + f_{1,2}^*(2) - \pi_1 - \pi_2 > f_{1,2,3,4,5}^*(1) + f_{1,2,3,4,5}^*(2) + f_{1,2,3,4,5}^*(3) + f_{1,2,3,4,5}^*(4) + f_{1,2,3,4,5}^*(5) - \pi_1 - \pi_2 - \pi_3 - \pi_4 - \pi_5$$

which reduces to $f_{1,2}^*(1) + f_{1,2}^*(2) > f_{1,2,3,4,5}^*(1) + f_{1,2,3,4,5}^*(2) + f_{1,2,3,4,5}^*(3) + f_{1,2,3,4,5}^*(4) + f_{1,2,3,4,5}^*(5) - \pi_3 - \pi_4 - \pi_5$.

The first statement tells us that we want board 3 to have a small dual and add a lot to the manufacturing cost. In order to have the second statement be true, the amount that board

3 adds to the reduced cost, needs to be offset by the presence of boards 4 and 5. This means that boards 1 and 2 should be fairly similar, but board 3 must be very different, with boards 4 and 5 being similar to board 3. We also want boards 4 and 5 to have large duals, so they need to be in bad clusters. So we created five boards with six components each that fit those descriptions:

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| 1 | 50 | 1 | 1 | 20 | 1 | 0 |
| 2 | 20 | 3 | 2 | 50 | 1 | 50 |
| 3 | 1 | 30 | 100 | 1 | 100 | 1 |
| 4 | 1 | 5 | 100 | 40 | 100 | 1 |
| 5 | 1 | 50 | 100 | 50 | 100 | 1 |

We then ran the master problem with three clusters: {1,4}, {2,5}, {3} and a setup cost $\sigma=200$. The duals we got back were 0, 0, 468, 705, and 860 for boards 1, 2, 3, 4, and 5 respectively. And sure enough, when we plugged those values back into the initial formulas we found that the reduced cost of boards 1 and 2 by themselves was 303, the reduced cost of boards 1, 2 and 3 was 336, and the reduced cost of all five boards together was -782. This means that we cannot, unfortunately, prune a branch if the objective function value gets worse by adding a particular board, since the objective function value might eventually go back down. It would be interesting to see if we could show that all clusters of this type would be better without the initial offending board to begin with – i.e. would cluster {1,2,4,5} always be better than {1,2,3,4,5}? We unfortunately did not have time to tackle this in the class, but it might be an interesting point to look into.

So we have shown that we cannot prune the tree going down, but perhaps we can gain some insight about the future of a branch based on what is left to explore. If the reduced cost of a node is positive, and the lower (or best) bound on the contribution of all the boards left to examine does not outweigh the current reduced cost, then there is no way the branch will ever be negative and we can prune it. We define the potential of a node to be the lower bound on the reduced cost contribution of the remaining boards, or $\sum_{i\in R} \lfloor f_i^*(i) - \pi_i \rfloor$, where $R$ is the set of all remaining boards. Since this is a lower bound on the contribution of the remaining boards, if the current reduced cost plus the potential is not negative, then we can prune the current node. This will hopefully save us some time when we near the bottom of the tree. We attempted to implement this pruning scheme, but unfortunately were stymied by our lack of object-oriented programming knowledge, and we could not figure out a way to store all the data of the tree in Matlab. Here again, formulating the data structures proved to be the most difficult part of the implementation, which is itself the most difficult part of testing the idea.

**<u>Solution (of sorts)</u>**

With this notion of potential in hand, we are now prepared to understand the final solution, though it probably would have taken us another semester (or at least another half a semester) to arrive at it on our own. The final solution involved a slightly modified version of this potential. The potential we defined above assumes each board is manufactured according to its own optimal setup, which we know is almost guaranteed not to happen in an actual solution. So how do you take that into account? We could say that we must account for the cost of manufacturing all the remaining boards together, i.e. if we had three boards left to examine, the potential would be $f_{1,2,3}^*(1)+f_{1,2,3}^*(2)+f_{1,2,3}^*(3)-\pi_1-\pi_2-\pi_3$. But this is not quite right either, since perhaps boards 1 and 2 should be manufactured together, but you might not want to include board three. So we now define potential to be the minimum reduced cost of all combinations of boards left to examine, here:

$$\min\left\{\sum_{i=1}^{3}(f_{1,2,3}^*(i)-\pi_i);\ \sum_{i=1}^{2}(f_{1,2}^*(i)-\pi_i);\ \sum_{i=2}^{3}(f_{2,3}^*(i)-\pi_i);\ \sum_{i=1,3}(f_{1,3}^*(i)-\pi_i),\ f_i^*(i)-\pi_i, i=1,2,3\right\}$$

In other words, we find the most negative reduced cost column among all the boards that we have not yet examined, and use that as our potential. Using this strategy, we can build all the potentials up from the bottom, using the potential from the previous level to help us. Then, by the time we reach the top, we have already determined the most negative reduced cost cluster. Indeed, we have found all the negative reduced cost clusters and we just need to move down the tree to find them. This method is a form of enumeration, very similar to what you see in dynamic programming where you perform backwards recursion to arrive at the beginning, at which point you have already solved the problem, but still need to traverse the tree forwards to explicitly state the solution. The final solution uses this technique, coupled with the pruning described earlier, to produce 2500 negative reduced cost columns at a time from the subproblem. Adding so many columns slows the solving of the master problem, but not by as much as shuffling back and forth between the master and subproblem 2500 times. And low and behold, it solves fairly quickly for nearly all instances up to 40 boards, with one glaring exception: the case where the optimal solution is the exhaustive board, i.e. when σ is very large.

**Return to Degeneracy**

Why would the algorithm take so very long to find a solution that it already had to begin with? (All instances started with an initial set of columns consisting of the identity, the exhaustive set, all 2 board clusters, all 3 board clusters, all n-2 board clusters, and all n-3 board clusters.) The answer belongs to our old friend, degeneracy. In a situation where the optimal solution is the exhaustive set, you have n-1 degenerate variables with value zero and 1 variable (the exhaustive set) with value one. The algorithm will continue to find negative reduced cost clusters, because it thinks it can make the objective value better, except all those negative reduced cost columns enter into the basis with value zero, and so have no effect on the objective value. There are nearly as many combinations of n-1 boards as there are of n boards, and you can cycle through most of them without seeing any change in the objective. The problem is that you have to find those degenerate columns and pivot them in, otherwise you will never know that you are optimal, because there will still be negative reduced cost columns out there. So how can you avoid this seemingly endless degenerate pivoting? You can't, basically, without exploiting some sort of problem structure.

In this instance, we noticed that the cost of the exhaustive column is $f_{all}^*(all)+\sigma$, if it is not optimal then we need at least two clusters, which means that our cost then will be at least 2σ plus some manufacturing cost. In order to make two columns optimal, we must be able to save something; since σ is increasing, the savings must come from the manufacturing cost. Thus, if $f_{all}^*(all)<\sigma$ then we know we are optimal because the savings cannot possibly outweigh the increase in setup cost. In the final version, in order to avoid the pain of endless degenerate pivots, a step was added at the beginning of the code to check for exactly that scenario, avoiding the problem altogether.

**Are we there yet?**

So we have a solution, or more rather, a way to obtain a solution. Was that not the point of the class? Well, we have a way to find a solution to the root node of the master problem, but what if the root node is fractional? Then you have to branch (and bound) to get an integer solution. Surprisingly, branching within the confines of column generation turns out to be quite challenging. The traditional way to branch takes a fractional variable, *x*, and says it has to be less than or equal to $\lfloor x \rfloor$, or greater than or equal to $\lceil x \rceil$, since it cannot be in between those two integer values. So we start one branch with $x = \lfloor x \rfloor$ and another with $x = \lceil x \rceil$, and we solve those and repeat the same procedure with any fractional variables in those solutions. The problem arises because we depend on the duals from the master problem to help solve the subproblem, and the duals come from the constraints. So if you add an additional constraint, say $x_1 = 0$, then the dual value for $x_1$ will be artificially low, and the dual for the added constraint will be very high, since there is great incentive to violate that constraint (otherwise the initial solution would not have been fractional). When you go to find the most negative reduced cost column, it will include that dummy dual from the added constraint, making the column you do not want to include the most negative reduced cost column. In fact, the column you want is the *second* most negative reduced cost column. In the next iteration, you will want the third most negative reduced cost column, and so on and so forth. This way of branching is simply untenable given the needs of our subproblem. So how do we branch?

Branching in this problem turns out to be much simpler than one might initially think. If we have a cluster, say boards 2 and 3, that is fractional, instead of setting $x_{2,3}$ to be zero or one, we can define the branching in terms of "togetherness." If we want to set $x_{2,3}$ to one, another way of saying that is we always want boards 2 and 3 together, so whenever we add board 2, we automatically add board 3 in with it. Continuing in this fashion, we can create a togetherness web—board 1 is with board 4, boards 2 and 3 are with board 7, board 5 is not with 4, etc. And this will eventually allow us to determine our integer solution without violating the needs of the subproblem.

## So what have we learned?

It has been a long, crazy journey on the road to this solution. But in this journey as in most, what is most important is not the destination, but the path taken to arrive there and the experiences gained along the way. The two most important things I learned from this course were the importance of thinking flexibly about problems, and that implementation is a pain. We explored the depths of frustration, we mounted the pinnacles of success, and saw that research is truly an up and down process—some of the things you try work out nicely, most of them will not; and you've got to learn to roll with the punches.

On the more practical side, this course really gave us a firm handle on what exactly column generation is and what sorts of problems we can adapt to fit it. Column generation is a massively powerful technique, yet elegant in its simplicity. I very much enjoyed getting my hands dirty with the nitty gritty of it all. We learned about degeneracy, and what a pain that can be. We learned about LP relaxations and how important a good formulation can be if you ever want to solve your problem. We saw that big *M*'s are almost always bad news, which will definitely encourage me to look for other ways to model my problems from here on out. One of the things that will really help me in the future is the ability to look at a fractional solution, and think about why it is fractional, step-by-step. Because many of the problems you model as integer programs will, in fact, turn out to be fractional, and many of them will make no sense as to why they are fractional. Being able to analyze a problem in the detail we did in this course, I think will help me a great deal in my future research.

We also saw how to exploit problem structure, and what great advantages that can give you. From now on, I will know to always look for symmetry in my models, to see if there is a way I can reduce the number of variables, and thus the complexity of my problem. We learned about branching—what had formerly seemed like a fairly straightforward process is now even more of a mystery. I can't say that I gained any deep insight into branching strategies, except that I more fully understand their vast complexity. I did learn, however, how to critically attack branching strategies, and the process one would go through to try and evaluate them, which I think will probably be useful in the future. We saw that using a programming language without good references can be frustrating in the extreme. The majority of my AMPL questions were answered not by the manual, but by other students in the department who had wrestled with these exact same questions.[1] But perhaps most importantly, we now have a whole toolbox full of skills to use when we are attacking a problem that just will not budge—you can take the dual of the problem and see if that gets you anywhere, you can try to reformulate it using entirely different variables, you can look to exploit problem structure, you can closely examine the branching strategy and see if you can gain any insight… Many of these approaches I never would have considered before taking this course. I feel like I am much better prepared to face the frustrations and difficulties that inevitably face researchers now. (Which is not to say I am *prepared*, per se, simply more prepared than I was previously.)

## What's Left

I really wish we had someone who knew how to write a .run file that would allow us to test the non-linear model. I would be very interested to see how it behaves and if, in fact, it would get us close enough to an optimal solution that then we could throw it at the symmetric model and prove optimality. There is no real way to know without writing that .run file, but it seems like it would be so simple, and the results so fascinating. I also feel like we had some interesting intuition about other branching strategies right there at the end that might have been interesting to explore. Not that it would have been particularly fruitful (given all the other complexities of the problem) but I would think with a little more work we might be able to prove the convexity of the optimal function; it definitely seems like it should be convex, and my dreams would be sweeter knowing that it was. Additionally, I would have liked to revisit the idea of genetic algorithms to see if we could concoct a heuristic using breeding that would get us close to the optimal solution. And while I am sure there could be multiple semesters taught on Dushant's local neighborhood search mechanisms, I would have enjoyed at least a little glimpse into

[1] It might make a fantastic project to pool all of the collective knowledge of the OR students and faculty and compile some sort of in-house AMPL ready reference – even just taking sample code from many students and highlighting the unique things the code does. I know this class (and my research) would have been made much easier had there been such a reference available.

that world. It also would have been interesting to look at the lagrangian relaxation—I do not know much about lagrangian relaxations, except what I saw in 518, Intro to Integer Programming, but they seemed to be powerful tools to help solve integer programs with nasty constraints. I feel like I finally have a handle on degeneracy for the first time in my life, which is rather exciting. That said, I still would have enjoyed looking at different situations where we can exploit problem structure to avoid degeneracy, since I feel that is one of the more important skills I gained from this class. People have talked about Bender's decomposition, but I had never even heard of Bender's decomposition until this class. Dantzig-Wolfe decomposition blew my mind—I hazard to guess what Bender's decomposition would have done to me.

And that, as they say, is all she wrote.

Aside: Amy, this was an absolutely wonderful class. I sincerely hope that you offer it again. I often looked forward to tackling the homework assignments from your class, which is exceedingly rare. You pushed us to think creatively and in ways I haven't been pushed to think in a long time. I cannot think of a more pleasant engaging way to learn many of these things, and in such a connected way, no less. I really and sincerely loved this class, and I can't wait to put these skills to use on other problems. So thanks. (There are times when I wonder if I could ever do this sort of stuff for the rest of my life, and if nothing else your class has confirmed for me that I really do enjoy it. It's easy to lose the excitement and joy of problem solving when you're slogging through problem set after problem set.)