Optimizing Product Line Designs:
Efficient Methods and Comparisons*

First Version: February 2005
Revised November 2005

Alexandre Belloni
belloni@mit.edu

Robert Freund
rfreund@mit.edu

Matthew Selove
mselove@mit.edu

Duncan Simester
simester@mit.edu

MIT Sloan School of Management
Cambridge MA 02142

* We thank Karen Freeman for research assistance on this project and Ely Dahan, John Hauser, and Olivier Toubia for the data used in the study. We thank Andreas Schulz for his insightful comments and for suggesting connections to the stable set problem in discrete optimization. This research has been partially supported through the Singapore-MIT Alliance.
Optimizing Product Line Designs:  
Efficient Methods and Comparisons

We compare a broad range of optimal product line design methods. The comparisons take advantage of recent advances that make it possible to identify the optimal solution of problems that are too large for complete enumeration. Several of the methods perform surprisingly well, including Simulated Annealing, Product-Swapping and Genetic Algorithms. The performance of the Product-Swapping heuristic is surprising given its simplicity.
1. Introduction

Soon after Green and Rao (1971) first proposed using conjoint analysis to measure customer preferences, academic researchers began studying how firms can use partworth estimates from conjoint studies to design optimal product lines. However, there is relatively little evidence validating the effectiveness of product line optimization methods on realistic problems. Previous validations have focused on small problems for which it is possible to identify the optimal design through complete enumeration. For larger problems, optimal solutions have been unobtainable, making it impossible to evaluate whether the methods yield solutions that are close to optimal. This limitation is important as the methods are designed for these larger problems in which complete enumeration is not possible. As a result, it has not been possible to evaluate the methods on the problems for which they were designed.

In this paper we compare the performance of a broad range of optimization methods on a more realistic product line design problem. The comparisons use conjoint data collected from real customers describing their preferences for computer laptop bags. We compare the quality of the resulting solutions, together with the difficulty of implementing the methods and the cost of computation. The comparisons are made possible by recent advances in optimization methods and computer hardware. Using a combination of fine-tuned discrete optimization methods that combine Lagrangian relaxation with branch-and-bound (hereafter referred to simply as Lagranian relaxation), we are able to compute guaranteed optimal solutions to realistic product line design problems that are too large for complete enumeration. We do not consider the Lagrangian relaxation method itself to be practical – it is extremely complicated and it is far too computationally intensive. Instead, we use the Lagrangian relaxation method to provide a benchmark for evaluating the solutions generated by other, more computationally efficient methods.

Although none of the methods (except for Lagrangian relaxation) guarantee global optimality, several of them consistently reach optimal solutions nonetheless. Even when measurement errors are introduced, the most successful methods still generate solutions
within 3% of the true optimal earnings. These findings suggest that realistic product line design problems might be easier to solve than previously thought. Surprisingly, for the problems we study, relatively simple methods are capable of computing near-optimal solutions with little computational burden.

Increases in computing power have increased the size of the problems that can be solved through complete enumeration. This trend is illustrated in Figure 1, where we plot the size of the problems solved in previous studies. We interpret a problem as “solved” if there is a guarantee that the solution is globally optimal. To measure problem size we use the logarithm (base 10) of the number of feasible product lines. For example, consider a product line of three different products that can vary on seven attributes, each of which have two possible levels. There are a total of 128 different products ($2^7$) and 341,376 unique combinations of three products. Complete enumeration would require that we evaluate all 341,376 product lines, and so our size measure is 5.53 (the log of 341,376).

**Figure 1. Size of Problems Solved in Previous Research**

The apparently linear increase in problem size between 1985 and 2003 corresponds to an exponential increase in the size of the problems (recall that our “problem size” measure is
logarithmic). In all of these papers optimality is guaranteed by complete enumeration and so this increase can be attributed solely to increases in computing power. By using recently developed methods that guarantee optimality without requiring complete enumeration, we can solve much larger problems. In particular, we consider two different sized problems in this paper. The “full” problem has almost 5,000 trillion feasible combinations, while the “truncated” problem has over 147 billion feasible combinations. If we could evaluate 30,000 product lines per second, it would take 57 days to solve the truncated problem through complete enumeration. For the full problem it would take over 5,000 years.

While the Lagrangian relaxation method computes an optimal solution as well as providing a guarantee of optimality, we do not consider it to be a practical solution method for solving product line design problems. The Lagrangian relaxation method is extremely complicated and computationally intensive. Solving the full problem took over a week of computation, while even the truncated problem required over 24 hours.

Although most of the previous evaluations of product line design methods consider problems that are small enough for complete enumeration, there are exceptions. For example, Nair, Thakur, and Wen (1995) and Balakrishnan, Gupta, and Jacob (2004) evaluate problems that are too large for complete enumeration. However, without knowing the global optimum, they can only compare the relative performance of the methods, without providing an absolute comparison to the true optimum. In practice, without knowing whether the methods produce results that are close to optimal, managers will find it difficult to evaluate whether it is worthwhile continuing to search for more profitable designs.

2. Product Line Design Methods

Product line design methods vary on a broad range of dimensions, including both the rules that consumers use to select products and the firm’s objective function. In this section we review these dimensions and characterize the methods evaluated in this study.
Consumer Choice Rules
The most common choice rule used in the product line optimization literature is first choice, which assumes that customers purchase the products that provide the most utility (calculated using the estimated partworths). Another common choice rule is the logit model, which assumes that products that provide a higher level of utility have a higher chance of being purchased, but that all available products have some positive probability of being chosen by each consumer. Sawtooth (2003) argues that both approaches have drawbacks. First choice tends to exaggerate the market share of popular products, while underestimating the share of unpopular products. Logit corrects for this problem, but exaggerates the market share of similar products. For example, if there are two products on the market that are exactly the same, the logit model would assign each of them the same probability of being purchased, so that their combined probability of purchase is nearly twice what it would be if there were only one product of the given type on the market.

Sawtooth (2003) suggests a third method know as randomized first choice, which estimates market shares by running multiple iterations that add a random error term to each partworth and a random error term to each overall product utility for each individual, and assuming that each consumer purchases the product that generates the highest utility after these random perturbations. This method avoids the drawbacks of both the first choice and logit models, but at the cost of significantly increased computation time.

In this paper we use the more common first choice rule, but the methods we test could be modified to use a logit rule or randomized first choice rule.

Objective Function
Zufryden (1977) first formulated the conjoint-based optimization problem to maximize market share. Green and Krieger (1985) introduced two other objective functions, known as the “buyer’s welfare problem,” which seeks to maximize the utility of consumers, and the “seller’s welfare problem,” which seeks to maximize profits for the firm. Kohli and
Sukumar (1990) provide mathematical formulations of these three problems, all of which are NP hard.\(^1\)

It is generally recognized that the seller’s welfare (profit maximization) problem is, in some respects, the most difficult of the three to solve. The share-of-choices problem is equivalent to a simplified version of the seller’s welfare problem in which all possible product types are equally profitable. McBride and Zufryden (1988) point out that the buyer’s welfare problem is also equivalent to a simplified form of the seller’s welfare problem.\(^2\) They find that integer programming can be used to solve the buyer’s problem, but not the seller’s problem. Intuitively, the difficulty of solving the seller’s problem arises from the fundamentally opposite objectives of customers and the firm, since customers want products with more features and lower prices, while firms prefer to sell products with fewer features and higher prices.\(^3\)

In addition to being the most general and difficult to solve, the seller’s welfare problem is often the most relevant of the three criteria. For this reason, we adopt this criterion in our comparisons.

### 3. Description of the Methods

We compare nine optimization methods, which can be grouped into three broad categories:

1. Methods that operate in attribute space.
2. Methods that operate in product space.
3. Methods that evaluate partially-formed products.

\(^1\) Kohli and Krishnamurti (1989) provide a detailed proof that the market share problem is NP hard. Kohli and Sukumar (1990) sketch a proof for the other two problems in a footnote on page 1469 of their paper.

\(^2\) As explained by Kohli and Sukumar (1990), the buyer’s welfare problem is equivalent to the seller’s welfare problem under the assumption that the profit that the firm derives from each attribute is equal to the utility that the consumer derives from that attribute. This means that the profit derived from an attribute varies by customer, as is true in the general formulation of the seller’s problem.

\(^3\) This intuition can be translated into an interesting mathematical condition. The share-of-choices problem and the buyer’s welfare problem can be cast as a maximization of a submodular function while the seller’s welfare cannot. We refer to Nemhauser et al (1978) for properties of submodular functions.
In this section we provide a brief description of each of these methods. At the end of the section we also briefly describe other methods and explain why it was not appropriate to include them in this comparison.

3.1. Methods That Operate In Attribute Space

Methods in this category begin by choosing a random solution (or set of solutions) and measuring the earnings level associated with this initial solution. The methods seek to improve the current solution by changing one or more product attributes, and then testing the impact that this change has on earnings. The following methods are included in this category.

**Coordinate Ascent**

This is our name for the method described in Green, Krieger, and Zelnio (1989). The coordinate ascent method begins by choosing a random product line and evaluating the profitability of this solution. The method then cycles through each product feature in a randomly chosen order, testing every possible level of each feature, and accepting feature changes that improve earnings while rejecting those that do not. These iterations continue until they no longer yield an improvement in earnings. The simple “one opt” version of this algorithm only tests a single feature change at a time. We also implement “two-opt” and “three-opt” versions of the algorithm, which simultaneously test two and three feature changes at a time.

The coordinate ascent method is guaranteed to find a locally optimal solution, where the “neighborhood” of locality is defined to include all solutions that differ from the current solution by a single feature (or by two features if using two-opt; and by three features if using three-opt). However, the method is not guaranteed to find a globally optimal solution. In particular, there is no guarantee that other solutions, which differ by more features, will not improve upon the current solution.
**Genetic Algorithm**

The biological process of natural selection provided the original inspiration for genetic algorithms. Genetic algorithms have been applied to a wide variety of problems in the operations research literature and were first applied to the optimal product design problem by Balakrishnan and Jacob (1996). Alexouda and Paparizzos (2001), Steiner and Hruschka (2003), and Balakrishnan, Gupta, and Jacob (2004) have also used genetic algorithms on product line design problems. Instead of beginning with a single random solution, genetic algorithms start with a population of random solutions. The “fittest” members of this initial population survive and move on to produce the next generation of solutions. New solutions enter the population through a process of reproduction (in which product lines “mate” to produce offspring) and mutation (in which product lines undergo random changes to individual product features). This process continues until a given stopping condition is reached, which in our implementation occurs when the entire population is homogenous (all product lines in the population are the same). Additional implementation details are provided in the Appendix.

Genetic algorithms are not guaranteed to find the global optimum solution. They are also not guaranteed to find a local optimum, but it is highly likely that they will, since the mutation process will usually identify any potential local improvements. Balakrishnan and Jacob (1996) suggest that genetic algorithms might be expected to perform well because they search for solutions from a number of different points in the solution space, increasing the odds of finding good solutions.

**Simulated Annealing**

Simulated annealing is a popular “algorithm of last resort” for difficult discrete optimization problems. The name of the method is derived from the physical process of annealing, in which a liquid is slowly cooled in a heat bath in order to form a solid in a low-energy state. A detailed introduction to simulated annealing can be found in Aarts, Korst, and van Laarhoven (1997). As far as we know, simulated annealing has not previously been applied to the optimal product line design problem.
Simulated annealing is similar to coordinate ascent in that it starts with a randomly chosen solution and proceeds to test random feature changes to the current solution. The difference is that the simulated annealing algorithm sometimes accepts feature changes that reduce earnings. The probability of accepting such a negative change depends on the magnitude of the drop in earnings and also decreases over time as the algorithm progresses through a pre-set “cooling schedule.” See the Appendix for additional implementation details.

Like coordinate ascent, simulated annealing guarantees a local optimum but not the global optimum. Because simulated annealing sometimes accepts feature changes that reduce earnings, it has the ability to escape from a locally optimal solution in the hope of finding a better solution. For this reason, the method is expected to outperform coordinate ascent.

3.2. Methods That Operate In Product Space

Methods in this category differ from those in the previous category in one important respect: instead of searching for an optimal solution by changing product attributes, these methods work by changing entire products. In order for these methods to work, it must be feasible to enumerate across the range of possible products, which represents the number of possible combinations of attributes in a single product. For this reason, these methods are only practical when the number of different possible products is not unreasonably large. The following methods are included in this category.

**Greedy Heuristic**

This method was first applied to the optimal product line design problem by Green and Krieger (1985), and has also been used (with some modifications) by Dobson and Kalish (1993) and Steiner and Hruschka (2003). The greedy heuristic begins by creating a product line that includes only one product, selected as the single product that maximizes earnings. It then proceeds to add one product at a time to the product line, always choosing the product that maximizes earnings given the set of products that have already been selected. The method stops when the desired number of products has been reached.
This method is not guaranteed to find a local optimum, since, for example, the first product added to the product line might not be locally optimal given the set of products that are subsequently added.

**Divide and Conquer Hueristic**

Green and Krieger (1993) suggest applying a “divide and conquer” heuristic to the optimal product line design problem. This method works by dividing the product line into two groups of attributes and completely enumerating all possible combinations for one group while holding the other group fixed. The groups are alternated until no further improvement is possible. For the problems we test, enumerating all possible values for one half of the attributes would require over five million product line evaluations, which is computationally prohibitive. In order to implement the heuristic more quickly, we break the product line into more than two groups, treating each product as its own “group” of attributes. Thus, we start with a random product line, and then optimize the choice of the first product, holding all other products constant, and then move on to the second product, and so on. This process continues until it is impossible to improve earnings by changing any single product.

This heuristic is guaranteed to find a locally optimal solution, with the local neighborhood defined to include all solutions that differ from the current solution by a single product. The resulting local neighborhood is larger than the local neighborhood defined in the one-opt coordinate ascent heuristic, giving the divide and conquer heuristic the ability to continue finding better solutions after reaching a point that would be considered a local optimum by one-opt coordinate ascent. For this reason, the divide and conquer heuristic might be expected to outperform one-opt coordinate ascent. Nevertheless, like coordinate ascent, the method is not guaranteed to find the global optimum. Because they can simultaneously change attributes of two or three different products, the local neighborhoods for the two-opt and three-opt coordinate ascent methods are neither strict subsets nor strict supersets of the neighborhood for the divide and conquer heuristic. Therefore, it is difficult to predict the relative performance of these methods.
We also considered an alternative implementation of the divide and conquer heuristic, which grouped the product features into three groups. The first group comprised the price feature alone; the second group included three non-price features; and the final group included the remaining three non-price features. Because this alternative implementation took significantly longer to run and produced less profitable solutions, we did not further explore this implementation.

**Product-Swapping Heuristic**

This is our name for a method that is similar to what Green and Krieger (1985) call the “interchange heuristic.” The product-swapping heuristic begins by choosing a random product line and evaluating the earnings level produced by this solution. It then tests each candidate product that is not part of the current solution to see if there is a product in the current solution whose replacement by the candidate product will increase earnings. If such a swap does improve earnings, then the candidate product is added, and the current product is removed from the current solution. The choice of which product to remove is made to achieve the maximum possible increase in earnings. This process continues until it is impossible to improve earnings by swapping in any single product. Green and Krieger (1985) suggest implementing the interchange heuristic by beginning with the solution from the greedy heuristic. However, we have found that beginning with a random solution works just as well, and has the advantage of making it possible to find multiple local optima by starting with different solutions.

Like the divide and conquer heuristic, the product-swapping heuristic is guaranteed to find a local optimum, with the local neighborhood defined to include all solutions that differ from the current solution by a single product.

**3.3 Methods That Evaluate Partially-Formed Products**

Whereas methods in the first two categories evaluate products, methods in this third category initially consider only a subset of product features, evaluating “partially-formed” products in order to eliminate certain combinations of features from
consideration. These methods are designed specifically for problems in which the number of possible product types is so large that their enumeration is computationally prohibitive. For problems of this scale, it is not practical to implement methods that operate in product space. Moreover, although methods that operate in attribute space could be implemented, their computational cost would become significantly larger.

We include this final category of methods in our analysis only for completeness. In the problems that we study it is possible to enumerate the possible product types, so the problem setting does not favor these methods. Furthermore, focusing on problems for which these methods were designed would prevent us from evaluating several of the other methods.

**Dynamic Programming Heuristic**

Kohli and Krishnamurti (1987) developed a dynamic programming heuristic to solve the optimal product design problem for a single product, which was subsequently extended to handle multiple products by Kohli and Sukumar (1990). This dynamic programming heuristic works by building the product line one attribute at a time. For example, consider a problem in which the first product attribute has seven possible levels, and the second attribute has two possible levels. The first stage of the heuristic evaluates each of the seven possible levels of the first attribute in terms of their impact both on consumer utilities and on marginal product profitability. The second stage would consider all fourteen possible combinations of attributes one and two. In order to prevent the number of attribute combinations from becoming too large, the heuristic eliminates certain combinations from consideration whenever the number of profiles in which a new attribute level appears exceeds the number of products in the product line, maintaining those combinations that appear most likely to produce a profitable product line. This process continues until the final attribute is reached and the number of attribute combinations that remains equals the number of desired products in the product line. For implementation details, see Kohli and Sukumar (1990).

This heuristic is not guaranteed to reach a local optimum. Note also that the outcome of this algorithm will depend on the order in which the attributes are added to the product
line. As suggested by Kohli and Sukumar (1990), we implement the heuristic for a number of different random attribute orderings. Each iteration of the DP heuristic is actually the maximum of ten iterations with random attribute orderings.

**Beam Search Heuristic**

Beam search methods were originally developed for Artificial Intelligence search problems in speech and image recognition. Nair, Thakur, and Wen (1995) were the first to apply these methods to the optimal product line design problem. The beam search heuristic is similar to the dynamic programming heuristic (described above) in that it evaluates various attribute combinations, retaining only those that seem likely to produce a profitable product line. There are several major differences between beam search and the dynamic programming heuristic. First, instead of proceeding one attribute at a time, beam search works by simultaneously combining different sets of attributes. For example, in the first stage of the heuristic, it might combine attribute one with attribute two, attribute three with attribute four, and so on. Second, instead of simultaneously creating an entire product line, beam search adds only one product to the product line at a time, in a manner similar to the greedy heuristic. Third, the techniques use different criteria for deciding which attribute combinations to eliminate at each stage. However, we have found that the beam search finds better solutions when using the criterion proposed by Kohli and Sukumar (1990), which we implement for both methods.5

---

4 Balakrishnan, Gupta, and Jacob (2004) also implement the beam search and develop a method for combining beam search with genetic algorithms.

5 Kohli and Sukumar (1990) describe their criterion for deciding which attribute combinations to maintain at the bottom of page 1470 of their paper. This process works in a similar manner to the greedy heuristic. For example, suppose that the most recently added feature is size; that there are 14 attribute combinations with the size attribute set to “large;” and that we wish to maintain only 5 of these combinations. The heuristic proceeds as follows. For each customer, calculate the incremental utility provided by each attribute combination relative to the customer’s status quo product. (For example, if the combination under consideration has the color attribute set to “black,” and the customer’s status quo product is “red,” calculate the customer’s incremental utility from receiving a black instead of a red product.) If an attribute combination provides positive incremental utility, calculate the earnings that the firm would generate by selling a product with that combination of attributes. Choose the attribute combination that yields the maximum total earnings across all consumers. (In our implementation, we assume the status quo product is sold by a competing firm, so the focal firm receives no profits from consumers who maintain the status quo.) Next choose the second attribute combination to maintain in a similar manner, taking into account that some customers would prefer the first combination that was selected, while others would prefer the new combination, depending on which provides higher utility. Continue in this manner until 5 attribute combinations have been selected. Nair, Thakur, and Wen (1995) devise an alternative method for deciding which attribute combinations to maintain, which they describe on page 774 of their paper. This approach
Like the dynamic programming heuristic, the beam search heuristic is not guaranteed to identify a locally optimal solution, and the outcome of beam search also depends on the system of pairing different attribute combinations at each stage. Nair, Thakur, and Wen (1995) find that a “best-worst” pairing method works slightly better than random pairings, but for the problem we test it is not clear how to define the “best” combination, given that one of the combinations includes price. As with the dynamic programming heuristic, we implement the beam search using different random orderings of attributes.

**Nested Partitions Heuristic**

Shi, Olafsson, and Chen (2001) apply a “nested partitions” heuristic to the product line design problem. This method works by dividing the solution space into various feasible regions, estimating which region appears most promising, and then sub-dividing the most promising region into smaller regions for further exploration. At each step of the heuristic, if the surrounding region, which includes all solutions not in the current most promising region, is estimated to be more promising than any of the regions under consideration, the heuristic backtracks to a larger super-region. We refer the reader to Shi and Olafsson (2000) and Shi, Olafsson, and Chen (2001) for implementation details.

The nested partitions method can use other product line design heuristics to help calculate the promising index of a region, in order to determine which region appears most promising. Shi, Olafsson, and Chen (2001) use a combination of genetic algorithms and a method they refer to as a greedy heuristic (which is different than the greedy heuristic we implement) to calculate each region’s promising index. To make the method run faster, we use the one-opt coordinate ascent method, taking the maximum of three iterations to calculate each region’s promising index. The nested partitions method can also use different rules for deciding how far to backtrack when the surrounding region is

---

seeks to reduce the computational and storage requirements of the heuristic by combining data on consumer part-worths and seller returns into a single matrix. For example, if an attribute combination provides positive incremental utility to a consumer and positive returns to the seller, the heuristic multiplies these two values together, producing a value which “serves as a surrogate for the expected margin before a complete product is defined.” We find that the beam search produces significantly better solutions when it instead uses the criterion proposed by Kohli and Sukumar (1990) to decide which attribute combinations to maintain, so we use this criterion in both the dynamic programming heuristic and the beam search heuristic.
more promising than the regions under consideration. We backtrack one level when this happens. We stop the method when a complete product line has been produced.

3.4 Guaranteed Optimality Benchmark

All methods discussed so far are heuristic methods that have been designed with the aim of generating good solutions without an excessive amount of computation. However, none of the methods are guaranteed to compute the globally optimal solution. Indeed, none of the methods can confirm global optimality even if they find the global optimum. Unfortunately, proving that a given solution is optimal for the product line design problem is hard (the problem is “NP-hard” in the language of theoretical computer science, Kohli and Krishnamurti 1989).

Until recently, the only method available to guarantee optimality was complete enumeration. However, we are now able to compute a guaranteed globally optimal solution to relatively large product line design problems using a fine-tuned advanced version of Lagrangian relaxation combined with a branch-and-bound method. This method is described in detail in the Appendix. While useful as a method to benchmark the performance of other solution methods, the computation time for the Lagrangian relaxation method makes it unlikely that it would be practical for most firms to implement: the Lagrangian relaxation method takes about 24 hours to solve the truncated version of the problem we study, and about seven days to solve the full problem.

Roughly speaking, the Lagrangian relaxation method is a two-step procedure that combines dual methods with primal branch-and-bound. The first step consists of an application of a Lagrangian relaxation variant called relax-and-cut as in Lucena (1992) applied to the product line design problem with additional customized valid inequalities for our specific implementation. To efficiently deal with the huge cardinality of the constraints induced by the consumers’ preferences, we dualize only a dynamic subset of the constraints at each outer iteration instead of dualizing all of the consumer preference constraints at once. This subset of constraints is enlarged or reduced heuristically in order
to keep its size small enough to manage, but large enough to enforce the constraint inequalities that are likely to be active at an optimal solution.

The second step of the Lagrangian relaxation method is a branch-and-bound procedure that solves the primal problem. At any point in this procedure we have current upper and lower bounds on the value of the optimal solution. We then branch the problem into two sub-problems where we fix one of the binary variables to be 1 in one sub-problem and 0 in the other. We next obtain bounds for the sub-problems. We keep branching until the upper bound becomes less than or equal to the lower bound (and we then cut this branch), or there is no further variable that is non-integer. The success of any branch and bound scheme relies on the quality of the bounds and on how fast such bounds are achieved. Due to the particular structure of the problem, the Lagrangian relaxation scheme is used to define the sub-problems at each node of the branch-and-bound tree. Also, the so-called root node is constructed based on the best solution found by the first step. For details of the implementation, we refer readers to the Appendix, where we provide a detailed description of the algorithm.

3.5 Other Methods

Our review of the literature identified several other methods for solving product line design problems. As we will discuss below, it was either not appropriate or not possible to include these methods in our comparison.

Dobson and Kalish (1988) develop a heuristic to design a near-optimal product line assuming that prices are continuous. Although the assumption of continuous pricing makes the problem more realistic, it also makes computation time for finding solutions much longer. To simplify computation Dobson and Kalish propose a two-stage optimization process, in which products designs are chosen first and optimal prices are then set.
McBride and Zufryden (1988) develop a linear programming method for finding a
globally optimal solution to the product line design problem, but they find that their
method is computationally prohibitive for the types of problems that we investigate.

Multidimensional scaling (MDS) methods have also been used to design product lines.
MDS assumes that each consumer has an ideal point in attribute space and that the utility
derived from a given product depends on its distance from that ideal point. In contrast,
conjoint-based methods assume that each consumer has a partworth for each potential
value of each product attribute, and that the utility derived from a product is the sum of
the partworths associated with the product’s attribute levels. Although both approaches
have their advantages, because they rely on different data sources, MDS-based methods
cannot easily be compared with conjoint-based methods. For this reason, we restrict
attention to optimization methods designed for use with conjoint analysis.

4. Design of Experiment

The problem that we use to compare the different methods was an actual product line
design problem faced by Timbuk2 (www.timbuk2.com), an innovative manufacturer of
messenger bags and other lifestyle items. Timbuk2 wanted to introduce a new laptop
computer bag. Although its products had traditionally been distributed through its own
website, where customers could customize the design of their bags, the company planned
to offer this new product for sale through traditional retailers.

As part of its product development efforts, Timbuk2 worked with an academic research
team to develop a conjoint study to evaluate customer preferences for different features
of a laptop computer bag. The conjoint study focused on price and nine other product
features. Each feature (including price) had two levels. The subjects were first-year
MBA students. An email invitation to 360 students yielded a total of 324 complete
responses, with each subject providing 16 paired-comparison responses. Additional
details are reported in Toubia, Simester, Hauser and Dahan (2003).
Toubia et al. used the conjoint study to validate an adaptive method of designing conjoint questions. In the study customers were randomly assigned to one of four groups, in which the design of the conjoint questions varied. In this study we pooled the raw data across the four groups and compared the optimization methods using data for 324 respondents. For each respondent we estimated individual partworths for each product feature using OLS. Because these design and estimation decisions are the same for each of the optimization methods, the results of our analyses should not be sensitive to these decisions.

We assume that Timbuk2 wants to design a product line with five products. For the price feature, we consider 7 different price levels ($70, $75, $80, $85, $90, $95, $100). Because the conjoint data only include two price levels ($70 and $100) we interpolate to derive partworths for the intermediate levels. The combination of nine features plus price yields a total of 3,584 unique bags ($2^9 \times 7$). There are over $4.9 \times 10^{15}$ different combinations of five-bag product lines that could be chosen from this set of 3,584 bags. Even using the sophisticated Lagrangian relaxation method that we adopt in this paper, it takes over one week of computation to compute the guaranteed optimal solution.

Unfortunately these computational requirements make it impractical to test the robustness of the solution by repeating the analysis on perturbed samples of data. For this reason, we initially focus on a truncated version of the problem in which we limit the binary design attributes to just six of the nine product features (the six features were chosen randomly). The combination of six two-level product features and a seven-level price feature yields 448 unique bags ($2^6 \times 7$). There are approximately $1.5 \times 10^{11}$ different combinations of five-bag product lines resulting from this set of 448 different bags. Our implementation of the Lagrangian relaxation method solves this truncated problem in approximately 24-hours. We later also evaluate the methods on the full problem; however computational limitations preclude performing robustness checks on the full problem.
Timbuk2 provided estimates of the cost of each feature. To make the problem more realistic we also included a selection of three competing bags. These competing bags were arbitrarily designed to include: a “fully loaded” bag with all six features priced at $100, a bag with three of the features priced at $85, and a “stripped down” bag with no special features priced at $70. We assumed that if the competing bags offered the same utility as one of the five bags in the Timbuk2 product line then the customer would purchase the competing bag. Notice that this assumption, together with the other elements of the experimental design, is common to all of the product line design methods that we evaluate. We also assumed that the competitor could not respond to Timbuk2’s product line design. We discuss this last issue further in our review of the limitations and opportunities for future research.

Although the number of feasible product lines is very large for both of the problems we study, the range of individual products is relatively small. This has some important implications for our evaluation of the optimization methods. First, as explained in the previous section, several of the methods we test can only be implemented on problems in which one can quickly enumerate all possible products, as is the case with our problems. In addition, we take advantage of the small number of possible products to speed up the performance of the methods by pre-computing (and storing) the matrix of customer utilities for each product (3,584 x 324 elements for the full version of the problem and 448 x 324 elements for the truncated version). We also pre-compute the profit margins that the firm earns from each product (3,584 elements for the full version and 448 elements for the truncated version). Pre-computing these values significantly reduces the computations that the methods must perform for subsequent product line evaluations. For both the full and truncated problems, the computational and memory requirements of these pre-computing tasks are trivial, but for other problems these tasks might be costly or prohibitive because the number of possible product types increases exponentially with the number of attributes in each product. As we observed earlier, the methods that evaluate partially formed products are specifically designed for these alternative settings.
5. Tournament Results

**Truncated Problem**

Table 1 presents results for ten trials of each optimization method on the truncated problem. In this initial analysis we assume that the partworths are measured without error and so we interpret them as the “true” partworths. We later investigate what happens when we add (simulated) measurement errors to these partworths. For each method, the table reports measures describing the average earnings, computational intensity, and the difficulty of implementing the different methods.

Accuracy is measured by the earnings under each of the ten trials (presented in descending order), together with the sample averages and standard deviations (where appropriate) for each method. We also report whether each method guarantees that the solution from each trial is locally optimal. In interpreting this guarantee, it is important to recall that the definition of neighborhood of locality varies across the methods. For methods that operate in attribute space, the neighborhood of locality is defined to include all solutions that differ from the current solution by a single feature change (in a single product). In contrast, for methods that operate in product space, the local neighborhood is defined to include all solutions that differ from the current solution by a single product.

Computational intensity is measured by both the CPU time used in each trial and the number of product lines evaluated in each trial. The CPU time was measured while running the methods in Matlab on an IBM Thinkpad laptop with a 1.7-GHz Pentium processor and 512 MB of RAM. To describe the difficulty of implementing the methods we report our subjective assessment of relative difficulty. In general, the methods we label as having a “medium” or “high” level of difficulty require some problem-specific fine-tuning of parameter values, while those we label as “low” do not.
### Table 1. Initial Comparison of Methods on the Truncated Problem

<table>
<thead>
<tr>
<th>Computed Earnings</th>
<th>Lagrangian Relaxation Branch &amp; Bound</th>
<th>Attribute Space Methods</th>
<th>Product Space Methods</th>
<th>Partially Formed Products</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Coordinate Ascent (One Opt)</td>
<td>Genetic Algorithm</td>
<td>Simulated Annealing</td>
</tr>
<tr>
<td>Trial 1</td>
<td>13,739</td>
<td>13,659</td>
<td>13,739</td>
<td>13,739</td>
</tr>
<tr>
<td>Trial 2</td>
<td>13,604</td>
<td>13,739</td>
<td>13,739</td>
<td>13,739</td>
</tr>
<tr>
<td>Trial 3</td>
<td>13,557</td>
<td>13,739</td>
<td>13,739</td>
<td>13,739</td>
</tr>
<tr>
<td>Trial 4</td>
<td>13,487</td>
<td>13,739</td>
<td>13,739</td>
<td>13,739</td>
</tr>
<tr>
<td>Trial 5</td>
<td>13,277</td>
<td>13,739</td>
<td>13,739</td>
<td>13,739</td>
</tr>
<tr>
<td>Trial 6</td>
<td>13,114</td>
<td>13,739</td>
<td>13,739</td>
<td>13,739</td>
</tr>
<tr>
<td>Trial 7</td>
<td>13,089</td>
<td>13,739</td>
<td>13,739</td>
<td>13,686</td>
</tr>
<tr>
<td>Trial 8</td>
<td>13,055</td>
<td>13,739</td>
<td>13,739</td>
<td>13,686</td>
</tr>
<tr>
<td>Trial 9</td>
<td>12,995</td>
<td>13,701</td>
<td>13,739</td>
<td>13,686</td>
</tr>
<tr>
<td>Trial 10</td>
<td>12,369</td>
<td>13,659</td>
<td>13,739</td>
<td>13,659</td>
</tr>
<tr>
<td>Average earnings</td>
<td>13,739</td>
<td>13,220</td>
<td>13,727</td>
<td>13,701</td>
</tr>
<tr>
<td>Percentage of optimal</td>
<td>100.0%</td>
<td>96.2%</td>
<td>99.9%</td>
<td>100.0%</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>389</td>
<td>27</td>
<td>12</td>
<td>30</td>
</tr>
<tr>
<td>Guaranteed local optima</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Computational Intensity</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average CPU Time</td>
<td>0.1 sec</td>
<td>10.3 sec</td>
<td>145.8 sec</td>
<td>1.3 sec</td>
</tr>
<tr>
<td>Total Evaluations</td>
<td>562</td>
<td>29,000</td>
<td>290,000</td>
<td>5,825</td>
</tr>
<tr>
<td>Evaluations per second</td>
<td>5,615</td>
<td>2,816</td>
<td>1,989</td>
<td>4,481</td>
</tr>
<tr>
<td>Subjective Difficulty</td>
<td>Very High</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</td>
</tr>
</tbody>
</table>
Average Earnings

There are several findings of interest. The earnings level of $13,739 determined by the Lagrangian relaxation method is the globally optimal earnings of the problem. This solution turns out to be unique (there is only one combination of bags that will yield this solution). Amongst the more practical methods, the genetic algorithm, simulated annealing, and product swapping perform best, reaching the optimal solution in more than half of their trials. The divide and conquer heuristic also performed very well, yielding average outcomes within 0.2% of optimal.

The greedy heuristic is the only method (other than Lagrangian relaxation) for which performance does not vary across the trials. Recall that this method begins by finding the single most profitable product. Conditional on this choice, it then finds the next most profitable product, and so on. Because each of these sub-tasks has a unique solution and does not vary due to any stochastic element, the outcome of repeated trials also does not vary. This unique outcome was $13,401, which was just $338 (2.5%) less than the optimal outcome.

Recall that the three methods that evaluate partially formed products (beam search, the DP heuristic, and nested partitions) were designed for problems in which it is computationally impractical to enumerate all possible product types. In our setting enumeration of product types is possible: the six product features and price yield just 448 different combinations of features in this truncated problem (3,584 in the full problem). Nonetheless, all three of the methods performed well, generally reaching a solution that was within 2-3% of the optimal design.

Coordinate ascent had the lowest average earnings and did not find the optimal outcome in any of the ten trials. The average earnings for this method were almost 4% less than the optimal solution. While the differences in average earnings might appear small, it is important to remember that these numbers do not account for fixed costs. If profit margins net of fixed costs are low, then small differences in gross earnings could imply substantial differences in net earnings. For example, if fixed costs are $12,000, then net
earnings when using the coordinate ascent are on average almost 30% lower than the earnings under simulated annealing and product swapping.

The findings reported in Table 1 for coordinate ascent are for the “one opt” version of the method. Recall that there are more powerful versions, which evaluate whether varying more than just a single feature can improve upon the current solution. We also implemented “two-opt” and “three-opt” versions, which simultaneously vary two and three features (respectively). A summary of the findings is reported in Table 1b below. As expected, increasing the number of features that can be changed at each evaluation increases the accuracy of the method, but also increases the computational intensity. We also see that there are decreasing returns to the additional computational time, due in part to a ceiling effect. Moving from one-opt to two-opt leads to a large jump in accuracy (from 96.2% to 98.7%) but only a modest increase in computational burden. In contrast, moving from two-opt to three-opt leads to only a modest gain in accuracy (98.7% to 99.8%) but a large increase in computation time.

There have been previous comparisons of selected pairs of these methods. Reassuringly, the findings reported in Table 1 are consistent with these comparisons. For example, Balakrishnan and Jacob (1996) present results comparing genetic algorithms and the DP heuristic. Their findings also favor the genetic algorithm. Similarly, Alexouda and Paparizzos (2001) find that genetic algorithms outperform beam search, while Steiner and Hruschka (2003) report that genetic algorithms outperform the greedy heuristic.

| Table 1b. Comparison of Coordinate Ascent Versions |
|----------------------------------|-----------------|-----------------|-----------------|
|                                  | One-Opt         | Two-Opt         | Three-Opt       |
| Average earnings                | 13,220          | 13,562          | 13,707          |
| Percentage of optimal           | 96.2%           | 98.7%           | 99.8%           |
| Standard deviation              | 389             | 192             | 37              |
| Average CPU run time            | 0.1 sec         | 3.2 sec         | 137.3 sec       |
| Total evaluations               | 562             | 11,373          | 552,151         |
| Evaluations per second          | 5,615           | 3,554           | 4,021           |
| Subjective difficulty           | Low             | Low             | Low             |
To help evaluate these findings, we also investigated two random strategies for choosing bags. In the first random strategy we randomly chose five bags from the set of 448 possible bags. The average earnings level reached in one hundred random solutions was only $8,071 (58.7% of the optimum). In the second random strategy we first chose three bags to undercut the competitors and then randomly selected the other two bags. To undercut the competitors we either matched the features and undercut the price, or matched the price and added an additional feature. One hundred repetitions of this strategy yielded average earnings of $9,458 (68.8% of the optimum).

Computation Time
There is wide variation in the computational intensiveness of the methods, with CPU times ranging from 0.1 seconds to almost 3 minutes. In general, the more computationally intensive a method is, the better it performs. We summarize this relationship in Figure 2, where we present a scatter plot of the average accuracy (earnings) and the average running time. The outliers to this trend are divide and conquer and product-swapping, which quickly achieve the maximum possible earnings using a relatively small number of evaluations. In general, the methods that operate in product space tend to outperform other methods with similar running times.

Caveats
These results should be interpreted with a number of caveats. First, several of the methods require some problem-specific fine-tuning of parameter values and other aspects of the algorithms. While we have tuned each method to this problem, it is possible that alternative implementations would lead to superior performance. Moreover, methods that perform poorly on average but exhibit a large standard deviation in earnings can do well if they are run a sufficiently large number of times and the best outcome is used. To

---

6 Matlab is designed to be particularly efficient with matrix operations, and hence algorithms that can be coded using matrix algebra run relatively efficiently in Matlab. However, algorithms that are iterative and run ordinary loops run less efficiently in Matlab. As a result, the number of evaluations per second for each method is influenced by the extent to which we were able to implement the method using matrix operations. For example, we implement the beam search almost entirely using matrix addition and multiplication, and this method performs an average 13,453 evaluations per second, much higher than any of the other methods. For this reason, the average number of evaluations performed might be considered a more meaningful measure of computational intensiveness than average running time.
illustrate this point, we ran 1,000 additional iterations of the one-opt heuristic, which had the worst average performance of all the methods we tested. This method reached the true optimum in 22 of these trials, or 2.2% of the time.

**Figure 2. Average Earnings versus Running Time**

![Figure 2. Average Earnings versus Running Time](image)

**Robustness Testing**

Our focus on the practical performance of the methods raises a natural question: “how well do the methods accommodate measurement error?” Although none of the methods were explicitly designed to account for measurement error, measurement error is generally unavoidable in practice. To test the robustness of the methods in the presence of measurement error we repeated our analysis after perturbing the original partworth estimates for the 324 respondents. These perturbations were accomplished by adding (simulated) error to the partworths:

\[ u'_{i,j} = u_{i,j} + \varepsilon_{i,j} \]
where $u_{i,j}$ is the original partworth for respondent $i$ on product feature $j$, $\epsilon_{i,j}$ is a zero-mean normally distributed perturbation term and $u'_{i,j}$ is the perturbed partworth. The standard deviation of the perturbations was obtained by using the standard errors for the respective $u_{i,j}$ terms. These standard errors were provided by the regression equations used to estimate each $u_{i,j}$. We repeated this process 20 times, obtaining 20 sets of perturbed partworths for each respondent (in addition to the original partworths).

These perturbed partworths offer two approaches for evaluating the robustness of our findings. Under the first approach, we can interpret the perturbed partworths as new values for the “true” partworths. We can then use the 20 sets of perturbed partworths to evaluate how robust our findings in Table 1 are to changes in the true partworths. Under the second approach, we can treat the perturbation terms as measurement error introduced by the data collection and estimation procedures. Under this interpretation, there is only one set of “true” partworths: the original partworths that we analyzed in Table 1. However, we assume that the researcher does not observe these true partworths and instead observes a set of sample partworths that are subject to measurement error. We have 20 sets of these sample partworths for each respondent. This approach allows us to evaluate how well each of the methods performs when the conjoint data on which they rely is subject to measurement error.

**Approach 1: Perturbed Partworths Are The “True” Partworths**

In Table 2 we compare how well the methods performed on the 20 sets of new “true” partworths. We implemented the Lagrangian relaxation method separately on each of the 20 new sets of partworths, which provided guaranteed optimal solutions for all 20 new datasets. Because each implementation took approximately 24 hours this was a very lengthy procedure. This also makes it clear why we study a truncated version of the problem: performing this robustness check on the full problem would have taken approximately 140 days.
Because the optimal earnings vary for each dataset, the “Percentage of Optimal” metric is calculated for each dataset and then averaged across the 20 datasets. Similarly, the “Average Rank” ranks the earnings of the methods on each dataset (where the most profitable method receives a rank of “1”) and then averages this rank across datasets. The average optimal earnings under the 20 perturbed datasets is $13,520. This is lower than the earnings under the original partworths ($13,739), which is an interesting and unexpected result. Our intuition for this outcome is that the perturbations increase the heterogeneity of the 324 respondents. This is confirmed by calculating the variance of the partworths across the 324 respondents: the variance of the partworths is higher in the perturbed datasets than in the original dataset. As the level of heterogeneity increases, it becomes more difficult to satisfy a large number of customers with a small product line. When heterogeneity is extreme each respondent prefers a different bag. At the other extreme, if all customers were homogeneous, then they would all prefer the same bag.

<table>
<thead>
<tr>
<th>Method</th>
<th>Average Earnings</th>
<th>Percentage of Optimal</th>
<th>Average Rank</th>
<th>Standard Deviation of Earnings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lagrangian Relaxation with Brand &amp; Bound</td>
<td>13,520</td>
<td>-</td>
<td>-</td>
<td>203</td>
</tr>
<tr>
<td>Coordinate Ascent (One-Opt)</td>
<td>12,826</td>
<td>94.9%</td>
<td>9.4</td>
<td>402</td>
</tr>
<tr>
<td>Coordinate Ascent (Two-Opt)</td>
<td>13,302</td>
<td>98.4%</td>
<td>5.6</td>
<td>300</td>
</tr>
<tr>
<td>Coordinate Ascent (Three-Opt)</td>
<td>13,468</td>
<td>99.6%</td>
<td>2.4</td>
<td>191</td>
</tr>
<tr>
<td>Genetic Algorithm</td>
<td>13,503</td>
<td>99.9%</td>
<td>1.8</td>
<td>215</td>
</tr>
<tr>
<td>Simulated Annealing</td>
<td>13,520</td>
<td>100.0%</td>
<td>1.0</td>
<td>203</td>
</tr>
<tr>
<td>Divide &amp; Conquer</td>
<td>13,472</td>
<td>99.6%</td>
<td>2.7</td>
<td>250</td>
</tr>
<tr>
<td>Greedy Heuristic</td>
<td>13,148</td>
<td>97.3%</td>
<td>7.9</td>
<td>223</td>
</tr>
<tr>
<td>Product-Swapping</td>
<td>13,463</td>
<td>99.6%</td>
<td>3.4</td>
<td>202</td>
</tr>
<tr>
<td>DP Heuristic</td>
<td>13,042</td>
<td>96.5%</td>
<td>9.1</td>
<td>202</td>
</tr>
<tr>
<td>Beam Search</td>
<td>13,219</td>
<td>97.8%</td>
<td>7.0</td>
<td>302</td>
</tr>
<tr>
<td>Nested Partitions</td>
<td>13,089</td>
<td>96.8%</td>
<td>8.5</td>
<td>276</td>
</tr>
</tbody>
</table>
Reassuringly, the other findings in Table 2 confirm the robustness of our earlier findings. The methods that performed well on the original dataset (Table 1) also perform well on the perturbed datasets. Simulated annealing, product-swapping, divide and conquer and the genetic algorithm have average earnings within 1% of the optimum, while the other algorithms perform less well.

**Approach 2: Perturbations Represent Measurement Error**

In this second approach, we assume that the original data are consumers’ true partworths, and that the 20 perturbed datasets represent measured partworths with measurement error equal to the perturbations. The optimization methods are applied to the perturbed datasets, and so they optimize over data containing measurement error. The “optimal” product lines computed using the perturbed data are then evaluated on the original data containing the “true” partworths. In this manner, we can compare the sensitivity of the methods to measurement error. Our findings are reported in Table 3. The “Average Earnings” and “Average Rank” are calculated in a similar manner as in Table 2. However, in this case the earnings and rank are evaluated using the original (unperturbed) data. Once again, the methods that performed well on the original dataset (Table 1) also perform well in this robustness check. Simulated annealing, product-swapping, divide and conquer, and the genetic algorithm have average earnings within 3% of the true optimum. Most methods now perform 2% to 4% worse than they do when we assume the dataset is 100% accurate. This degradation reflects the loss of information due to measurement error.

**The Full Problem**

To investigate how the findings are affected if we increase the size of the problem we also compared the methods on the full problem. Recall that in the full problem the menu of features includes nine binary product features and price. The change from six to nine binary product features increases the number of product lines by over four orders of magnitude. Table 4 reports the results for ten trials of each method on this larger problem.
Table 3. Second Robustness Check:  
Perturbations Represent Measurement Errors

<table>
<thead>
<tr>
<th>Method</th>
<th>Average Earnings</th>
<th>Percentage of Optimal</th>
<th>Average Rank</th>
<th>Standard Deviation of Earnings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lagrangian Relaxation with Brand &amp; Bound</td>
<td>13,739</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Coordinate Ascent (One-Opt)</td>
<td>12,706</td>
<td>92.5%</td>
<td>9.9</td>
<td>502</td>
</tr>
<tr>
<td>Coordinate Ascent (Two-Opt)</td>
<td>13,209</td>
<td>96.1%</td>
<td>5.4</td>
<td>440</td>
</tr>
<tr>
<td>Coordinate Ascent (Three-Opt)</td>
<td>13,414</td>
<td>97.6%</td>
<td>2.8</td>
<td>185</td>
</tr>
<tr>
<td>Genetic Algorithm</td>
<td>13,441</td>
<td>97.8%</td>
<td>2.9</td>
<td>157</td>
</tr>
<tr>
<td>Simulated Annealing</td>
<td>13,455</td>
<td>97.9%</td>
<td>2.4</td>
<td>147</td>
</tr>
<tr>
<td>Divide &amp; Conquer</td>
<td>13,480</td>
<td>98.1%</td>
<td>1.9</td>
<td>130</td>
</tr>
<tr>
<td>Greedy Heuristic</td>
<td>13,237</td>
<td>96.3%</td>
<td>7.0</td>
<td>137</td>
</tr>
<tr>
<td>Product-Swapping</td>
<td>13,323</td>
<td>97.0%</td>
<td>4.4</td>
<td>225</td>
</tr>
<tr>
<td>DP Heuristic</td>
<td>13,111</td>
<td>95.4%</td>
<td>8.2</td>
<td>260</td>
</tr>
<tr>
<td>Beam Search</td>
<td>13,292</td>
<td>96.7%</td>
<td>6.0</td>
<td>175</td>
</tr>
<tr>
<td>Nested Partitions</td>
<td>13,161</td>
<td>95.8%</td>
<td>7.7</td>
<td>209</td>
</tr>
</tbody>
</table>

Note: Lagrangian Relaxation was run on the original dataset (with no assumed measurement error).

There are again several findings of interest. Lagrangian relaxation confirms that there is again a unique optimal solution to the problem, whose optimized earnings is $12,226. Simulated annealing was the only method in our comparison that consistently found the optimal solution, though it was again one of the most computationally expensive of the methods. This performance was closely followed by product swapping, for which the average earnings were just $6 lower than optimal. The relative performance of the other methods compared to simulated annealing remained largely unchanged from the truncated version of the problem, although there is a small drop in the relative performance of some of the less accurate methods.

With one exception (simulated annealing) the methods all took longer to solve this problem than they did for the truncated version of the problem. This reflects both an increase in the number of mathematical operations required to evaluate each product line, and an increase in the number of feasible product lines. Further investigation revealed
the reason that simulated annealing takes a similar time to run on the full and truncated problems. Price is more frequently selected as the feature to change in the truncated problem, and it takes slightly longer to randomly change the price than to change one of the non-price attributes.

We conclude that the robustness of the findings is reassuring. The pattern of results is relatively unchanged when we vary the size of the problem, introduce measurement error, or perturb the values of the true coefficients.

<table>
<thead>
<tr>
<th>Method</th>
<th>Average Earnings</th>
<th>Percentage of Optimal</th>
<th>Standard Deviation of Earnings</th>
<th>Avg. CPU Run Time</th>
<th>Total Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lagrangian Relaxation with Brand &amp; Bound</td>
<td>12,226</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Coordinate Ascent (One-Opt)</td>
<td>11,217</td>
<td>91.7%</td>
<td>691</td>
<td>0.2 sec</td>
<td>726</td>
</tr>
<tr>
<td>Coordinate Ascent (Two-Opt)</td>
<td>11,977</td>
<td>98.0%</td>
<td>97</td>
<td>5.4 sec</td>
<td>19,689</td>
</tr>
<tr>
<td>Coordinate Ascent (Three-Opt)</td>
<td>11,999</td>
<td>98.1%</td>
<td>85</td>
<td>302.8 sec</td>
<td>1,093,751</td>
</tr>
<tr>
<td>Genetic Algorithm</td>
<td>12,105</td>
<td>99.0%</td>
<td>112</td>
<td>16.5 sec</td>
<td>41,500</td>
</tr>
<tr>
<td>Simulated Annealing</td>
<td>12,226</td>
<td>100%</td>
<td>0</td>
<td>128.7 sec</td>
<td>290,000</td>
</tr>
<tr>
<td>Divide &amp; Conquer</td>
<td>12,176</td>
<td>99.6%</td>
<td>80</td>
<td>12.5 sec</td>
<td>51,969</td>
</tr>
<tr>
<td>Greedy Heuristic</td>
<td>12,035</td>
<td>98.4%</td>
<td>-</td>
<td>3.5 sec</td>
<td>17,920</td>
</tr>
<tr>
<td>Product-Swapping</td>
<td>12,218</td>
<td>99.9%</td>
<td>3</td>
<td>14.1 sec</td>
<td>51,926</td>
</tr>
<tr>
<td>DP Heuristic</td>
<td>11,539</td>
<td>94.4%</td>
<td>294</td>
<td>5.5 sec</td>
<td>10,864</td>
</tr>
<tr>
<td>Beam Search</td>
<td>11,486</td>
<td>93.9%</td>
<td>564</td>
<td>1.9 sec</td>
<td>19,038</td>
</tr>
<tr>
<td>Nested Partitions</td>
<td>11,818</td>
<td>96.7%</td>
<td>174</td>
<td>8.4 sec</td>
<td>33,099</td>
</tr>
</tbody>
</table>

6. Conclusions

We have compared a broad range of optimal product line design methods on a realistic problem. The comparisons take advantage of recent advances that make it possible to compute a guaranteed optimal solution to problems that are too large for complete enumeration. Somewhat surprisingly, we find that it is relatively easy to generate near-
optimal solutions to the problems that we study. Four of the methods, simulated annealing, product-swapping, divide and conquer, and genetic algorithms perform particularly well, consistently reaching near optimal solutions even in the presence of measurement error.

The product-swapping heuristic is remarkable for its simplicity, yet it also consistently finds optimal or near-optimal solutions, thus defying the conventional wisdom that sophisticated algorithms outperform their simpler counterparts. In this respect, the product line design problem may share similar characteristics to other problems that are easier to solve in practice than indicated by computational theory. For example, the traveling salesman problem (see Gutin and Punnen 2002) and the knapsack problem (see Kellerer, Pferschy, and Pisinger 2004) are both theoretically hard combinatorial problems. Nevertheless, relatively simple heuristics have proven successful at solving large versions of these problems.

We caution that the findings are subject to several potential limitations. The first limitation is that our measure of accuracy presumes that the partworths accurately describe actual customer behavior. Although we were able to demonstrate that the results are robust to moderately large unbiased errors in the partworth estimates, they may not survive systematic biases in these estimates. Of course this limitation relates not just to optimization, but to conjoint analysis itself. If conjoint analysis provides inaccurate predictions of market share, then little information is learned from the analysis. If these biases can be predicted, it may be possible to develop a model that does describe actual behavior. For example, Kivetz, Netzer, and Srinivasan (2004) find that modifying conjoint analysis to incorporate a compromise effect significantly improves choice predictions. Further research would be required to investigate how such modifications to the objective function affect the relative performance of the different optimization methods.

A related issue is whether cost predictions are accurate enough to make optimal product line design meaningful. Srinivasan, Lovejoy, and Beach (1997) find that for some
product types a simple linear cost model leaves much of the variance in product cost unexplained. Again, if it were possible to develop a more realistic model of product cost, the optimization methods could be modified to use the improved predictions. However, it is not clear how well the optimization methods would perform on the modified problem. Some authors have also observed that some combinations of product features may not be feasible, and so product line design methods need to be able to accommodate constraints on the set of feasible product lines. For example, the DP heuristic includes a mechanism that explicitly accommodates infeasible alternatives. Fortunately this issue did not arise in this setting, as Timbuk2 could in practice produce a bag containing any combination of the nine binary features.

A third concern is that none of the optimization methodologies consider the competitive response to new product introductions. The optimal solutions all predict large market shares for the focal firm, which could reasonably be expected to provoke an aggressive competitive reaction. Dobson and Kalish (1988) develop an optimization methodology that accounts for price response by competitors, but at the cost of significantly increased computation times. Developing an optimization methodology that adequately accounts for competitive reaction and is tractable for large-scale design problems remains an important challenge for future research.

Finally, while we have been able to evaluate a product line design problem that is dramatically larger than those previously studied, the problem is still at the lower bound of problems that firms would like to solve in practice. For now, we do not know how well the simple methods we study in this paper would perform on larger problems. In time, as both the algorithms and computational power improve, we anticipate that the frontier of problems that can be evaluated will continue to expand.
References


Appendix.

I. Implementation Details of Algorithms

Genetic Algorithms

Our genetic algorithm implementation starts with a population of 500 randomly chosen product lines. The earnings level associated with each of these initial solutions is tested, and the 250 with the highest earnings levels are chosen to move on and create the next generation of solutions. Of these 250 survivors, 125 pairs are chosen at random (with replacement) to “mate” and produce a total of 250 offspring. The literature suggests a number of different genetic crossover methods to create these offspring. We implement the “single-point crossover,” meaning that all attributes beyond a randomly chosen crossover point are swapped. For example, a child might inherit the first product and the first five features of the second product from the “father,” and inherit the rest of the features of the second product and all other products from the “mother.” We find that this crossover method performs slightly better than uniform crossover, in which each attribute of each offspring has a fifty percent chance of taking on the value of each parent. After the process of reproduction takes place, each of the 500 current solutions (250 original solutions plus 250 offspring) is copied to produce 500 new solutions. These new solutions then undergo a process of mutation, in which each feature of each solution is changed with a random probability (we used a mutation rate of 5%). Finally, of the 1,000 total solutions now under consideration, the 500 with the highest earnings level move on to form the next generation. This process continues until a defined stopping condition is met, which in our implementation occurs when the entire population is homogeneous, that is, all 500 product lines are the same.

Simulated Annealing

Our implementation of simulated annealing was done following the guidelines in Aarts, Korst, and van Laarhoven (1997). Like the coordinate ascent algorithm, simulated annealing tests random feature changes and accepts all changes that increase earnings. The key feature of simulated annealing is that it also accepts some changes that reduce earnings. If a feature change reduces earnings, then the probability of accepting the change is given by the following exponential function:

\[ P(\text{Accept Change}) = \exp\left(\frac{\text{NewEarnings} - \text{OldEarnings}}{C}\right) \]

where \( C \) is a control parameter that is analogous to temperature in the physical annealing process. Note that the greater the drop in earnings, the lower the probability that the feature change will be accepted. Also, as the control parameter \( C \) (the “temperature”) drops, the probability of a given change being accepted drops as well.
The term “cooling schedule” refers to the choice of a set of values of the control parameter $C$ and the number of feature changes to test for each value. Although Aarts, Korst, and van Laarhoven (1997) offer some general guidelines, setting the cooling schedule is partly a matter of trial-and-error and problem-specific experience. We found that it was best to start with a value large enough that almost all feature changes are accepted, calculate each subsequent value by multiplying the current value by 0.8, and end with a value small enough that very few negative changes are accepted. We divided the problem into 29 time stages, and tested 10,000 feature changes in each stage, yielding a total of 290,000 tested feature changes. The control parameter was set at each stage according to the following schedule: 1443, 1154, 923, 739, 591, 473, 378, 303, 242, 194, 155, 124, 99, 79, 63, 51, 41, 32, 26, 21, 17, 13, 11, 9, 7, 5, 4, 3, 0.1.

II. Description of the Lagrangian relaxation/branch-and-bound method

Preliminaries

We present a detailed description of our Lagrangian relaxation/branch and-bound method for the product line design problem. This method can be used to generate guaranteed optimal solutions for problems where complete enumeration is computationally intractable. Recall that Lagrangian relaxation is generically designed to compute upper bounds on the (unknown) optimal objective function value (see for example Fisher 1981), thus providing a conservative estimate of how “good” any feasible solution really is. By construction, these upper bounds need to be relatively efficient to compute. Unfortunately, such efficiency comes at a price, since there is no guarantee that the upper bound will match the true optimal value. The difference between the optimal objective function value and the lowest upper bound obtained by Lagrangian relaxation is the “duality gap” in mathematical programming parlance.

The branch-and-bound method (see for example Bertsimas and Tsitsiklis 1997), can be used to solve a given problem to optimality. It uses a “divide and conquer” approach to explore the set of feasible solutions, which in our case is the set of all product lines. In order to avoid exhaustive enumeration of all feasible solutions, branch-and-bound uses upper and lower bounds on the optimal objective value to restrict the search to promising regions only. In our case, the upper bounds derived from Lagrangian relaxation will be used to restrict the search, as will heuristics that generate hopefully-good feasible product lines and their subsequent lower bounds. We emphasize that branch-and-bound and Lagrangian relaxation methods have been extensively used in the literature on a variety of discrete optimization applications, see for example Lucena (1992). However, one needs to carefully exploit the particular structure of the problem at hand in order to obtain a successful implementation for large-scale problems.
Mathematical Programming Formulation and Lagrangian Relaxation Problem

With this in mind, we now describe our mathematical programming formulation of the product line design problem. Let $J$ denote the set of all possible products and let $I$ denote the set of consumers. For each consumer $i \in I$, her preference is represented by a mapping $\prec_i$, where $j_2 \prec_i j_1$ means that consumer $i$ prefers $j_1$ over $j_2$. The conjoint model implies that all products can be linearly ordered by each consumer, or equivalently, that the preferences are transitive. Let $p_j$ denote the profit obtained from selling one unit of product $j$ for $j \in J$, and denote by $k$ the size of the product line to be designed ($k = 5$ in our problem instance). We associate a binary variable $x_{ij}$ with consumer $i$’s decision to buy product $j$ or not, and we associate another binary variable $y_j$ with the firm’s decision to produce product $j$ or not. The resulting mathematical programming formulation of the product line design problem is as follows:

\[
\text{(MP): } \max_{x,y} \sum_{j \in J} \sum_{i \in I} p_j x_{ij} \\
\text{s.t. } \sum_{j \in J} y_j \leq k \quad \text{(1.1) (capacity)} \\
\sum_{j \in J} x_{ij} \leq 1 \quad \text{for all } i \in I \quad \text{(1.2) (consumption)} \\
x_{ij} \leq y_j \quad \text{for all } (i,j) \in I \times J \quad \text{(1.3) (availability)} \\
x_{ij} \leq 1 - y_j \quad \text{for all } (i,j) \in I \times J, j \prec_i j' \quad \text{(1.4) (preferences)} \\
x_{ij} \in \{0,1\}, y_j \in \{0,1\} \quad \text{for all } (i,j) \in I \times J \quad \text{(1.5) (binary)}
\]

There are four types of constraints in the model: the firm’s capacity, consumers’ consumption, product availability, and consumers’ preferences. The capacity constraint (1.1) ensures that the size of the product line is at most $k$. The model ensures that each consumer will choose at most one product in (1.2). Moreover, consumers can choose product $j$ only if it is available, which is reflected in (1.3). Lastly, each consumer must choose according to her preferences among the existing products, captured by the preference inequalities (1.4). This model is general enough to capture all the relevant features of our problem instance, including $m$ competing products (simply introduce $m$ artificial products which are always available (the corresponding $y_j = 1$) and have zero profit, and use $k + m$ instead of $k$).

In order to define the Lagrangian relaxation, we need to construct a dual function based on (MP) which will generate upper bounds on the optimal value. The underlying idea is to remove some constraints from the model and instead introduce penalties into the objective function for violating those same constraints. Define the set
$Q := \left\{(x,y) \in \{0,1\}^{I \times J} : x_j \leq y_j, \sum_{j=J} y_j \leq k \right\} = \{(x,y) : (1.1), (1.3), \text{ and } (1.5) \text{ are satisfied}\}$

and nonnegative Lagrange multipliers $\lambda$ and $\sigma$ for the consumption constraints (1.2) and preference constraints (1.4), respectively. Now define the following dual function:

$$f(\sigma, \lambda) = \max_{(x,y) \in Q} \sum_{(i,j)} p_j x_j - \sum_{i \in I} \sigma_i \left( \sum_{j \in J} x_j - 1 \right) - \sum_{(i,j) \in \Gamma} \lambda_{ij} (x_j + y_j - 1).$$

(2)

Lagrangian relaxation is built around the observation that for every $\sigma \geq 0$ and $\lambda \geq 0$, $f(\sigma, \lambda)$ is an upper bound on the optimal value of (MP). To see this, consider any feasible solution $(x, y)$ of (MP). By construction, $(x, y) \in Q$ and the additional terms are nonnegative since $\sigma \geq 0$ and $\lambda \geq 0$. In order to obtain the sharpest upper bound, we attempt to solve the associated Lagrange dual problem:

$$\begin{align*}
\min_{\sigma, \lambda} & \quad f(\sigma, \lambda) \\
\text{s.t.} & \quad \sigma \geq 0, \lambda \geq 0.
\end{align*}$$

(3)

Since $f(\sigma, \lambda)$ is actually a convex function, this minimization can be efficiently implemented, at least in a theoretical sense.

**Computational Considerations of the Lagrangian Relaxation Problem**

Two fundamental conditions must be met for the Lagrangian relaxation approach to be sensible in practice: first, the dual function (2) must be easy to evaluate for any given $(\sigma, \lambda) \geq 0$, and second, the dual problem (3) must not be large-scale, that is, the number of Lagrange multipliers should not be excessively large, say, no larger than $10^6$. We now examine these two issues in detail. In order to evaluate the dual function $f(\sigma, \lambda)$, we need to exploit its particular structure. The key observation is that a consumer can buy many products in the relaxed problem (2). Thus, looking at the “Lagrangian profit” associated with the variable $x_j$ in (2) ($\ell_{ij} := p_j - \sigma_i - \sum_{j' \in J} \lambda_{ij'}$), it is optimal to set $x_j = 0$ if $\ell_{ij} < 0$.

On the other hand, if $\ell_{ij} \geq 0$, it is optimal to set $x_j = y_j$, that is, consumer $i$ buys all products with positive Lagrangian profits that are available. Therefore we can restate the dual function (2) as:

$$f(\sigma, \lambda) = \max \left\{ \sum_{j \in J} \left( \sum_{i \in I} \max \{ \ell_{ij}, 0 \} - \sum_{j' \in J} \lambda_{ij'} \right) y_j + \sum_{j \in J} \sum_{j' \in J} \lambda_{ij'} + \sum_{i \in I} \sigma_i \right\}$$

s.t. $\sum_{j \in J} y_j \leq k$, $y_j \in \{0,1\}$, $j \in J$.

(4)

For fixed $\sigma$ and $\lambda$, the evaluation of (4) is simply to choose the $k$ products associated with the $k$ largest coefficients $c_j = \left( \sum_{i \in I} \max \{ \ell_{ij}, 0 \} - \sum_{j' \in J} \lambda_{ij'} \right)$. 

As mentioned earlier, the value obtained in (3) usually does not equal the optimal value of (1). In order to obtain even better bounds, we construct “valid inequalities” for the model (1) and dualize them as we did with the consumption and preferences constraints. (We refer the reader to Nemhauser and Wolsey (1988) for methods for constructing and using valid inequalities.) Although such valid inequalities are redundant for (1), they are not redundant for the associated dual problem (3). We have developed three different families of valid inequalities specifically for our algorithmic setting:

**Enlarged preference inequalities:** for each $j \in J$, let $H(j) \subseteq \{(i,m): j \succ_i m\}$ be a set of consumer-product pairs, and let $N$ be the number of different consumers in $H(j)$. The following inequality is valid for (MP):

$$\sum_{(i,m) \in H(j)} x_{im} + Ny_j \leq N. \tag{5}$$

**Circular preference inequalities:** given three consumer-product pairs $\{(i_k, j_k)\}_{k=1}^3$ such that $j_2 \succ_h j_1 \succ_h j_3 \succ_i j_2$ holds, the following inequality is valid for (MP):

$$x_{ij_1} + x_{ij_2} + x_{ij_3} \leq 1. \tag{6}$$

**Exclusive preference inequalities:** for a subset of consumers $Z \subseteq I$, let $\{T_i\}_{i \in Z} \subseteq J \setminus \{j\}$, be a family of disjoint subsets of products such that

- for $a,b \in Z$, $k \succ_a m$ and $m \succ_b k$ for all $(m,k) \in T_a \times T_b$, and
- for $a \in Z$, $j \succ_a m$ for all $m \in T_a$.

Then the following inequality is valid for (MP):

$$y_j + \sum_{i \in Z} \sum_{k \in T_i} x_{ik} \leq 1. \tag{7}$$

Notice that the total number of dualized constraints is much bigger than $10^6$ (in fact, it is exponential in the number of consumers and products), contradicting one of the fundamental conditions mentioned earlier. To overcome this, we use a dynamic version of the traditional Lagrangian relaxation approach proposed by Lucena (1992) which considers only a small (dynamic) subset of the inequalities at each iteration. This dynamic strategy is particularly suitable to our problem context since we expect that only a very small subset of the constraints will be active at any optimal solution. The relaxed solutions and associated Lagrange multipliers are used to add and remove inequalities to the set of “active” dualized constraints in a manner that will be described shortly.

In our implementation, all of the consumption constraints are always actively dualized, whereas the preference constraints (1.4), enlarged preference (5), circular (6), and exclusive preference (7) constraints are dualized in a dynamic fashion. For the sake of exposition, we describe our general method using only the preference constraints and their Lagrange multipliers $\lambda$, instead of all of the inequalities. Denote by $D_t$ the set of dualized inequalities at iteration $t$ and let $z(\lambda) = (x(\lambda), y(\lambda))$ denote a solution that
achieves (2) for the given $\lambda \geq 0$. Note that a subgradient $s(\lambda)$ of the function $f(\lambda)$ at $\lambda$ is easily computed once $z(\lambda)$ is known (for details see Bonnans et al. 2003). Our algorithmic scheme for dynamically changing the set of dualized inequalities is as follows:

Step 1 (Initialization). Let $D_0 := \emptyset$, $\lambda^0 := 0$, and $t := 0$.
Step 2 (Evaluate the Function). Compute $f(\lambda)$, $z' = z(\lambda)$, and $s' = s(\lambda)$.
Step 3 (New Inequalities). Generate the set $A_i$ of new inequalities based on $z'$.
Step 4 (Update Multipliers). Update multipliers $\lambda$ to $\lambda^{t+1}$ based on $s'$.
Step 5 (Update Inequalities). Let $R_i \subseteq \{ i \in D_i : \lambda_i^{t+1} = 0 \}$ and define $D_{t+1} = D_t \cup A_i \setminus R_i$.
Step 6 (Stopping Criterion). Verify a stopping criterion.
Step 7 (Loop). Set $t \leftarrow t + 1$ and go to Step 2.

The convergence of this scheme to an optimal solution depends on the particular rules used to update the Lagrange multipliers, the rules to update the set of active inequalities $D_t$, and on the rules for the stopping criterion. Under certain regularity assumptions, convergence has been established when bundle methods are used to update the Lagrange multipliers and a specific rule is followed to manage the inequalities (for details see Belloni and Segastizábal 2004). Our implementation uses a variant of the subgradient optimization method (for example, see Fisher 1981) which was also used in Lucena (1992 and 1993), and Belloni and Lucena (2004).

One important step that bears more detailed discussion is how to generate new inequalities based on $(x(\lambda), y(\lambda))$. This is known as the separation problem in combinatorial optimization. In our case, due to the structure of the problem, the vector $y(\lambda)$ will have at most $k$ nonzero elements, which implies that at most $kI$ components of $x(\lambda)$ will be nonzero. This reduces the search for violated inequalities to a relatively small subset only, since any violated inequality must have nonzero variables. For example, for each consumer the search for a violated enlarged preference inequality with $N = 1$ consists of only $k$ evaluations of her preference for each of the products assigned to the product line (up to $k$ products). We developed elementary heuristics to search for violated circular preference and exclusive preference constraints. Again, the integrality of the relaxed solutions $(x(\lambda), y(\lambda))$ considerably simplifies the search.

The Branch-and-Bound Method.

We start by briefly reviewing branch-and-bound, and refer the more curious reader to Bertsimas and Tsitsiklis (1997) for a more complete description. Branch-and-bound starts with a feasible set and generates associated upper and lower bounds. It chooses to "branch" the current problem into two subproblems where a particular variable is fixed at 0 and 1 in each of the two subproblems, respectively. The method then proceeds recursively: for each subproblem, upper and lower bounds are computed, another variable is chosen to be fixed, and more branches are created. Note that the upper bound will decrease as more variables are fixed. Hopefully, the lower bounds will exceed the upper
bounds for many branches, thereby eliminating the need to search those branches any further.

In order to apply the branch-and-bound method to solve (MP), we need to define the so-called branching rule as well as the method to generate upper bounds. The Lagrangian relaxation method is the natural choice to generate upper bounds, since fixed variables can be easily incorporated into the subproblem. In our implementation no new equalities were inserted or deleted at any node, i.e., we kept the same set of inequalities used at the root node. On the other hand, we did update the Lagrange multipliers on certain nodes of the tree to take advantage of the impact of fixed variables. Furthermore, if exactly $k - 1$ variables were fixed at 1 in some subproblem, we then solved the subproblem exactly.

With respect to the branching rule, the “$x$” variables were never fixed and branched on our implementation, since fixing and then branching on “$y$” variables had a much larger impact on the upper bounds. The component $j$ chosen for branching was determined by computing the largest coefficient $c_j$. 