FORAGING-DIRECTED ADAPTIVE LINEAR PROGRAMMING: 
AN ALGORITHM FOR SOLVING NONLINEAR MIXED 
DISCRETE/CONTINUOUS DESIGN PROBLEMS

Kemper Lewis
Systems Realization Laboratory, 
School of Mechanical Engineering, 
Georgia Institute of Technology, 
Atlanta, Georgia 30332-0405, USA 
Tel: (404) 894-8170 Fax: (404) 894-9342  
kemper@srl.marc.gatech.edu

Farrokh Mistree
Systems Realization Laboratory, 
School of Mechanical Engineering, 
Georgia Institute of Technology, 
Atlanta, Georgia 30332-0405, USA 
Tel: (404) 894-8412 Fax: (404) 894-9342  
farrokh.mistree@me.gatech.edu

ABSTRACT
Design models often contain a combination of discrete, integer, and continuous variables. Previously, the Adaptive Linear Programming (ALP) Algorithm, which is based on sequential linearization, has been used to solve design models composed of continuous and Boolean variables. In this paper, we extend the ALP Algorithm using a discrete heuristic based on the analogy of an animal foraging for food. This algorithm for mixed discrete/continuous design problems integrates ALP and the foraging search and is called Foraging-directed Adaptive Linear Programming (FALP). Two design studies are presented to illustrate the effectiveness and behavior of the algorithm.

NOMENCLATURE
DSP Decision Support Problem 
ALP Adaptive Linear Programming 
FALP Foraging-directed Adaptive Linear Programming 
UB Upper Bound LB Lower Bound 
TV Target Value TS Tabu Search

1 FRAME OF REFERENCE
1.1 Mixed Discrete / Continuous Optimization in Design
Optimization techniques have become an integral part of a design process where values for system variables must be found subject to a set of constraints, bounds, and objectives. A recent review of the literature dealing with the use of optimization in design is presented in (Papalambros 1995) and therefore will not be repeated here. In many cases, the system variables in optimization problems are assumed to be continuous. But in design, this is not always the case. In a given design problem, there may exist system variables which are continuous, integer, or discrete. Examples of these are shaft lengths, number of gear teeth and gear diameters, respectively. There are well established methods for solving continuous problems. These are largely calculus-based and usually require evaluation of derivatives (e.g., gradient-based solvers). There are also well established methods for solving discrete problems. These are largely based on some heuristic (e.g., branch and bound, Genetic Algorithms, Simulated Annealing, Tabu Search), since unlike its continuous counterpart, optimality criteria such as the Karush-Kuhn-Tucker conditions for discrete problems do not exist. Mixed discrete/continuous problems present mathematical programming challenges from both the continuous and discrete domains. The solution of these mixed problems is identified in (Papalambros 1995) as being “one of the most daunting problems in design optimization.”

As a starting point, a general mixed discrete/continuous optimization problem is stated as follows:

Find X
\[ x_1, x_2, x_3, x_4, x_5 \]
\[ x_1 \leq x_2 \leq x_3 = X \]
\[ x_1 \leftrightarrow x_2 \leftrightarrow x_3 = 0 \]
Satisfying
Constraints

1 NASA Graduate Researcher, Associate Member ASME
2 Professor, ASME Fellow, Corresponding Author
where $X_C$ are the continuous domain variables, $X_I$ are the integer domain variables, and $X_D$ are the discrete domain variables. This general model encompasses all classes of mathematical models. There are different modeling techniques to convert the general model for its solution by different codes. Our philosophy for this conversion is detailed in (Mistree, Patel et al. 1994). This philosophy is encompassed in the compromise Decision Support Problem (DSP), a multiobjective decision model which incorporates concepts from both traditional mathematical programming and goal programming (Mistree, Hughes et al. 1993). In short, converting a general baseline model into a compromise DSP includes converting objectives into goals, establishing priority among the goals and representing the goals as equations by using deviation variables. We recognize that design problems are inherently multiobjective and optimizing with respect to each objective is impractical and many times impossible. Therefore, we approach the problem from a satisficer's perspective (Simon 1982). Let us explain. Consider a haystack with a number of needles hidden in it. An optimizer will continue to search the haystack until the last needle has been found. A satisficer, on the other hand, stops when he/she has found enough needles to proceed to the next step. The compromise DSP has the capability to model problems from both a satisficing and an optimizing perspective (Mistree, Patel et al. 1994). Our perspective in this paper is one of an optimizer. We use the compromise DSP to model mixed discrete/continuous optimization problems and focus on their single unique solution.

The new solution algorithm to solve mixed discrete/continuous design problems is called the Foraging-Adaptive Linear Programming (FALP) Algorithm. We illustrate this idea in Figure 1. We have labeled three primary constructs in the figure, A, B, and C. The ALP Algorithm, construct B in the lower half of Figure 1, uses gradients to move through the continuous design space. As part of the work in this paper, we have developed a search engine, construct A in the upper half of Figure 1, based on the notion of foraging of animals in the wild to intelligently search the discrete solution space for promising regions. In the foraging literature, foraging behavior is characterized by empirical observations and simple analytical models. Therefore, our foraging solver is not based on an accepted theory of foraging but on foraging behavior observed in animals. The foraging search is not constrained by the convexity of the design space, taking a higher-level perspective of the design space and using heuristics to search it, as shown in Figure 1. Information is passed from the discrete solver to the continuous solver using one common mathematical construct, the compromise DSP, construct C linking A and B in Figure 1. Therefore, we combine the a continuous solver (ALP) and discrete solver (foraging) into one algorithm (FALP) for mixed discrete/continuous problems. In the remainder of Section 1 we introduce the three fundamental constructs of Figure 1. In Section 2, we detail the step-wise FALP Algorithm, and in Section 3, we demonstrate the effectiveness of FALP using two well-studied examples.

### 1.2 The Compromise DSP: An Overview

The compromise DSP is a multiobjective decision model which is a hybrid formulation (Mistree, Hughes et al. 1993). It incorporates concepts from both traditional Mathematical Programming and Goal Programming. The compromise DSP is used to determine the values of design variables to satisfy a set of constraints and to achieve as closely as possible a set of conflicting goals. The compromise DSP is used to model such decisions since it is capable of handling constraints, goals, and multiple objectives (Mistree, Patel et al. 1994). In particular, the compromise DSP offers the following capabilities:

- accurately represent single-objective or multi-objectives
- use either preemptive or Archimedean formulation to prioritize objectives
- have hard constraints or soft constraints (goals)
- generate feasible solutions more frequently
- quickly generate results for several different weighting schemes

The system descriptors, namely, system and deviation variables, system constraints, system goals, bounds and the deviation function are described in detail elsewhere (Mistree, Hughes et al. 1993) and are therefore not be repeated here. The mathematical form of the compromise DSP is
summarized in Figure 2. In the compromise DSP, each goal, $A_i$, has two associated deviation variables $d_{i-}$ and $d_{i+}$, which indicate the extent of the deviation from the target. The deviation variables, $d_{i+}$ and $d_{i-}$, are both positive, and the product constraint, $d_{i+} \cdot d_{i-} = 0$, ensures that at least one of the deviation variables for a particular goal is always zero. If the problem is solved using a vertex solution scheme (as in the ALP-algorithm (Mistree, Hughes et al. 1993)), then this condition is automatically satisfied.

### Given

An alternative to be improved through modification.

Assumptions used to model the domain of interest.

The system parameters:

- $n$: number of continuous system variables
- $l$: number of discrete/integer system variables
- $p+q$: number of system constraints
- $p$: equality constraints
- $q$: inequality constraints
- $m$: number of system goals

#### System goals (linear, nonlinear)

$g_i(X) = 0 ; \quad i = 1, \ldots, p$

$g_i(X) \cdot 0 ; \quad i = p+1, \ldots, p+q$

#### System constraints (linear, nonlinear)

$A_i(X) + d_{i-} - d_{i+} = G_i ; \quad i = 1, \ldots, m$

### Find

$X_i \quad i = 1, \ldots, n+1$

$d_{i-}, d_{i+} \quad i = 1, \ldots, m$

### Satisfy

#### System constraints (linear, nonlinear)

$g_i(X) = 0 ; \quad i = 1, \ldots, p$

$g_i(X) \cdot 0 ; \quad i = p+1, \ldots, p+q$

#### System goals (linear, nonlinear)

$A_i(X) + d_{i-} - d_{i+} = G_i ; \quad i = 1, \ldots, m$

### Bounds

$X_i \min \cdot X_i \cdot X_i \max ; \quad i = 1, \ldots, n$

$d_{i-}, d_{i+} \cdot 0 ; \quad i = 1, \ldots, m$

$\left( d_{i-} \cdot d_{i+} = 0 ; \quad i = 1, \ldots, m \right)$

### Minimize

preemptive deviation function (lexicographic minimum)

$Z = \left[ f_1(d_{i-}, d_{i+}^+), \ldots, f_k(d_{i-}, d_{i+}^+) \right]$

---

### Figure 2. Mathematical Form of a Compromise DSP

Usually, goals are not equally important. To determine a solution on the basis of preference, the goals may be rank-ordered into priority levels. Customers rate certain product qualities higher than other qualities. Designers should seek a solution which minimizes all unwanted deviations from the desired qualities. There are various methods for measuring the effectiveness of the minimization of these unwanted deviations.

### 1.3 The ALP Algorithm: An Overview

The solution algorithm for continuous compromise DSPs is the ALP Algorithm. We have developed the Adaptive Linear Programming (ALP) Algorithm based on mathematical programming techniques to solve compromise DSPs. An overview of the ALP Algorithm is given here but the details are given in (Mistree, Hughes et al. 1993). Solution algorithms fall into two categories, namely,

- those that solve the exact problem approximately, and
- those that solve an approximation of the problem exactly.

Gradient-based methods, pattern search methods, and penalty function methods fall into the first category whereas methods involving sequential linearization fall into the second category. We chose the sequential linear programming approach in 1981 because it had, in our opinion, the highest potential for being used to develop a single algorithm for solving a range of DSPs in engineering design. More recently, Azarm et al. (Azarm, Dierolf et al. 1988) report that this is one of the most widely used approaches. We believe three important features contribute to the success of the ALP algorithm, namely,

- the use of second-order terms in linearization,
- the normalization of the constraints and goals and their transformation into generally well-behaved convex functions in the region of interest,
- an “intelligent” constraint suppression and accumulation scheme.

These features are described in detail in (Mistree, Hughes et al. 1981). Once the nonlinear compromise DSP is formulated, it is approximated by linearization. At each stage the solution of the linear programming problem is obtained by a Multiplex algorithm (Ignizio 1985). The deviation function that is given in the mathematical form of the template can be implemented in two ways:

1. In the Preemptive form as a lexicographic minimum of the goal deviation variables.
2. In an Archimedean form as a weighted function of the goal deviation variables.

The ALP Algorithm is only capable of handling continuous and Boolean variables. In this paper, we extend the ALP Algorithm using an intelligent search engine which facilitates the handling of discrete and integer variables. There are various other methods for solving mixed optimization problems, but this work is rooted in the notion of satisficing and is applicable to both satisficing and optimizing problems. In this paper, however, we illustrate only its applicability to optimization problems. The examples presented in Section 3 are single objective, and are used for comparison purposes. We believe design problems are inherently multi-objective, and application to future design problems will include multiple objectives. In the next section the final component, construct $A$, from Figure A, the foraging meta-heuristic is introduced.
1.4 The Notion of Foraging in Optimization

Our discrete solver, labeled A in Figure 1, is modeled after the natural process of foraging by animals in the wild. The foundation for this search is the Tabu Search (TS) (Glover 1989). We extend TS using constructs, which parallel the process of foraging by animals in the wild. The notion of using biological and evolutionary metaphors to model optimization has been explored and hypothesized upon in Glover (Glover and Greenberg 1989). In Figure 3, we illustrate TS and our foraging model using a simple cartoon.

In Figure 3, the rabbit (forager) begins the search by picking up a scent (in effect, the gradient). He moves in the direction of the strongest scent (largest descent gradient) until a solution (local optimum 1) is reached. But, the rabbit thinks that places with more food exists, so he continues his search. This takes him to another site with food (local optimum 2). The search continues to another site (local optimum 3). Here, if the forager were to simply look for the strongest scent, it would lead him back to local optimum 1. But with the memory structure, he remembers he has been there already, and continues the search elsewhere, eventually leading to the site with the most food (global optimum). The two fundamental developments of the foraging search in this paper are the use of a dynamic memory structure and schema identification. We define the parallel aspects of the foraging analogy to discrete optimization:

<table>
<thead>
<tr>
<th>FORAGING</th>
<th>DISCRETE OPTIMIZATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>areas with food</td>
<td>local optimum</td>
</tr>
<tr>
<td>area with most food</td>
<td>global optimum</td>
</tr>
<tr>
<td>search steps</td>
<td>discrete variables</td>
</tr>
<tr>
<td>regional boundaries</td>
<td>bounds</td>
</tr>
<tr>
<td>territorial boundaries</td>
<td>constraints</td>
</tr>
<tr>
<td>experience</td>
<td>memory structure</td>
</tr>
<tr>
<td>site</td>
<td>specific set of design</td>
</tr>
<tr>
<td>variables</td>
<td>region of allowable</td>
</tr>
<tr>
<td>neighborhood</td>
<td>dynamic memory</td>
</tr>
<tr>
<td>moves</td>
<td>schema identification</td>
</tr>
<tr>
<td>increasing hunger</td>
<td>objective:</td>
</tr>
<tr>
<td>characteristics of food areas</td>
<td>find best solution</td>
</tr>
<tr>
<td>objective:</td>
<td>in reasonable time</td>
</tr>
<tr>
<td>find most food in a small amount</td>
<td></td>
</tr>
<tr>
<td>of time</td>
<td></td>
</tr>
</tbody>
</table>

In this paper, the verification examples are strictly single objective, and have strictly been used as a means of comparison. Therefore, in this paper, we take an optimizer’s perspective. Our results are presented from an optimization standpoint, but we refer throughout the paper to instances where a satisficing standpoint is taken in developing the algorithm. A strength of our algorithm for mixed discrete/continuous design problems is the capability to handle multiple objectives. Although it has not been demonstrated in this paper, FALP uses a lexicographic routine to handle multiple objectives, which we feel are inherent in design problems.

In Section 2, we present the step-wise details of FALP, which is based on the three notions introduced in this section, namely foraging, the ALP Algorithm, and the compromise DSP. In Section 3, we present some results of applying FALP to mixed discrete/continuous design optimization problems.

2 FALP: THE MIXED DISCRETE/CONTINUOUS ALGORITHM

A schematic of the FALP algorithm is given in Figure 4. Implementation of the algorithm includes essentially 2 stages: the discrete solver (foraging), construct A, and the continuous solver (ALP), construct C. They are linked by the compromise DSP, construct B. Alone, each solver would be ineffective in solving a mixed discrete/continuous problem. But the integration and interaction of the solvers provides the basis for our mixed algorithm.
The basic step-wise solution procedure of FALP is summarized as follows:

**Step 1:** Initialize problem and parameters

**Step 2:** Solve the discrete domain, $X_D$ and $X_C \in D$, using foraging search.

**Step 2a:** Discretize the continuous variables.

**Step 2b:** Find best candidate solution in local neighborhood, $N(x)$.

**Step 2c:** Check dynamic memory list to allow/disallow solution.

**Step 2d:** Build new solutions or enact diversification scheme based on schema, if necessary.

**Step 2e:** Update list of best solutions encountered for user-interactive schema identification.

**Step 2f:** If maximum number of iterations is reached, select best solution visited. If not go to step 2a.

**Step 3:** Solve the continuous problem, $X_C \in R$, using ALP based on the information in Step 2.

**Step 3a:** Set discrete variables constant.

**Step 3b:** Construct Linear Problem using Sequential Linearization

**Step 3c:** Solve Linear Problem using Multiplex Algorithm

**Step 3d:** If continuous solution has converged, go on. If not, go to step 3b.

These steps are embodied within the three fundamental constructs of the algorithm in Figure 1. These steps are more closely discussed.

**Step 1:** Initialize problem and parameters

**Step 2:** Solve the discrete domain, $X_D$ and $X_C \in D$, using foraging search.

**Step 2a:** Discretize the continuous variables.

Previous versions of our algorithm involved the isolation of the discrete solver and the continuous solver. But, because of the decoupling of the two solvers, non-optimal solutions were found. The reasons for this are discussed in (Pan and Diaz 1990). Therefore, to alleviate this difficulty, the discrete and continuous solvers are coupled through the continuous variables. In the discrete solver, a discretized domain, $D$, is created for the continuous variables. Presently, the continuous variables are discretized using 10 discrete steps across the continuous domain specified from the lower and upper variables bounds. This low-fidelity discretization allows for exploration of the continuous domain without the expense of searching too fine of a discretization. Once the best neighborhood for the continuous variables is found, the continuous solver can refine the solution.

**Step 2b:** Find best candidate solution in local neighborhood, $N(x)$.

The basis for Tabu Search is described as follows (Glover 1989; Glover 1989; Bland and Dawson 1991). In general terms, TS is an iterative improvement procedure in that it starts from some initial solution and attempts to determine a better solution by applying a greatest-descent procedure. However, TS is characterized by a capability to escape local optima by using short and long term memory of visited solutions. Moreover, TS permits backtracking to a previous solutions, which may ultimately lead, via a different direction, to better solutions. The features of a tabu list and aspiration make TS a powerful optimization tool for models characterized by discrete variables (Bland and Dawson 1991; Ford and Bloebaum 1993). Given a set of objectives to be met over a set $X$, TS proceeds from one point in the design space to another until a chosen termination criterion is satisfied. Each $x \in X$ has an associated neighborhood $N(x) \neq X$, and each solution $x'$ $N(x)$ is reached from $x$ by an operation called a move. For the discrete variables, the neighborhood is easily defined. For the continuous variables, the neighborhood restriction is that $x'$ $N(x) \in D$. TS goes beyond local search by employing a strategy of modifying $N(x)$ as the search progresses, effectively replacing it by another neighborhood $N^*(x)$. A key aspect of TS is the use of special memory structures which serve to determine $N^*(x)$ and hence to organize the way in which the space is explored.

**Step 2e:** Check dynamic memory list to allow/disallow solution.

The solutions admitted by these memory structures are determined in several ways. One of these, which gives TS its
name, identifies solutions encountered over a specified \( N(x) \) and forbids them to belong to \( N^+(x) \) by classifying them tabu. The short term tabu lists in previous work have been assumed to be constant. Extending this static memory structure into a dynamic memory structure is one of our fundamental developments.

Researchers using the Tabu Search have assumed that the short and long term list lengths are constant. We feel allowing a changing list according to design space characteristics and search progress could generate better solutions more efficiently. This parallels the approach of Simulated Annealing (SA) where the probability of accepting a new move changes based on the progress of the search. This is also seen in the behavior in animals foraging for food. The use of dynamic memory is evident in animals foraging for food (Todd and Kacelnik 1993; Benhamou 1994). The tendency for an animal to continue searching early in the search process for better "finds" as opposed to later is frequently observed (Huntingford 1984; Todd and Kacelnik 1993) based on relative energy levels. The mathematical description of the dynamic memory is as follows, 

\[
\text{memory\_size} = \frac{1}{(\text{solution\_iteration})}
\]

where

\[
(\text{memory\_size}) \text{ is the length of forbidden (tabu) moves list.}
\]

In other words, the foraging search will accept worse solutions (longer list of forbidden moves) earlier in the solution process. As the process continues, the foraging search (rabbit) becomes more and more content (smaller list of forbidden moves) with the best solution found thus far. The dynamic memory memory structure is used in each neighborhood, \( N(x) \), to determine if a given design point is allowable. In other words, the search determines if the site has already been visited based on the present length of the memory list and then allows or disallows the move.

**Step 2d:** Enact diversification scheme based on stationarity of design variables, if necessary.

If certain variable values occur over a large number of iterations, then the algorithm enact a diversification scheme. The diversification scheme forces a change in the variable identified as having remained constant for a large number of iterations. Ideally, this diversification scheme would allow the algorithm to escape from local optima. This diversification scheme has also been used in versions of the TS.

**Step 2e:** Update list of best solutions encountered for user-interactive schema identification.

This notion parallels a similar notion in Genetic Algorithms (GA). That is, characteristics or schema (variable values) which occur frequently in the best solutions are identified and new solutions can be found based on this set of characteristics. In the foraging search, a record is kept of what discrete variable values occur as the solution of each iteration. Presently, the user can then "build" solutions using the schema as the foundation. This could greatly simplify the problem if schema are identified. With reference to Figure 3, note the flowers (\( \bigodot \)) that are present at most of the food sites. It has been observed in the behavior of many animals that they identify certain characteristics of places where food is found and will look for these guiding characteristics in determining future sites (Menzel 1991). Therefore, the Figure 3, the rabbit would ideally recognize the presence of the flowers and in the future identify the flower and then look for food in close proximity.

**Step 2f:** If maximum number of iterations is reached, select best solution visited. If not go to step 2a.

The foraging algorithm is a unassuming algorithm. That is, it will continue to search until a maximum number of iterations is reached. This is a very important aspect of the algorithm, stopping criteria. In order to ensure efficient design space search, in foraging, the maximum number of iterations is proportional to the problem size (number of variables and number of discrete values).

**Step 3:** Solve the continuous problem, \( X_C \in R \), using ALP based on the information in Step 2.

**Step 3a:** Set discrete variables constant.

The discrete variable values found in Step 1, \( X_D \), are set constant. The best continuous neighborhood, \( X_C \in N_C \) is given from Step 2.

Our continuous solver, labeled \( \bigcirc \) in Figure 1, is the Adaptive Linear Programming Algorithm (ALP) which is detailed in (Mistree, Hughes et al. 1993). This is not the focus of this paper, so only a brief description is given in Section 1.3.

**Step 3b:** Construct Linear Problem using Sequential Linearization

**Step 3c:** Solve Linear Problem using Multiplex Algorithm

**Step 3d:** If continuous solution has converged, go on. If not, go to step 3b.

In the next section, we illustrate the effectiveness of the algorithm using two example problems. These examples are single objective, and are used for comparison purposes. We believe design problems are inherently multi-objective, and application to future design problems will include multiple objectives.

### 3 VERIFICATION STUDIES

The two problems in this section have been well studied by other researchers. We have used them only as a means of verification, comparison, and illustration of FALP. Therefore, we use compromise DSPs, which are typically multiobjective, to model single objective optimization problems only to illustrate and verify the FALP algorithm. In the spring design problem, we illustrate and prove that the global optimum is found by FALP. We also illustrate the behavior of FALP using the spring design problem. With certain problems, globally optimal solutions may be found, as is demonstrated in the spring design problem. But with more complex design problems, finding these optimal solutions may not be feasible or possible.
In the pressure vessel design problem, we do not prove a global optimal condition, but illustrate that the solution is an improvement from the previous studies. We also illustrate how the previous studies can be improved upon using active constraint and monotonicity arguments.

3.1 Coil Compression Spring Design (Sandgren 1990; Kannan and Kramer 1994)

This is a problem involving discrete, integer, and continuous variables. A helical compression spring is to be designed as shown in Figure 5. The goal is to minimize the volume of the spring. The spring is to be manufactured from music wire spring steel ASTM A228. Therefore, the wire diameter can assume only the discrete values shown in Table 1. The design variables are D, the winding diameter (continuous), d, the wire diameter (discrete), and N, the number of spring coils (integer). The units for D and d are inches. The constraint is shear stress limit. The constraints , and are geometry limits. The constraint is to ensure proper winding. The constraints , , and are for deflection requirements.

Table 1. Possible Wire Diameter for ASTM A228 (inches)

<table>
<thead>
<tr>
<th></th>
<th>0.0090</th>
<th>0.0095</th>
<th>0.0104</th>
<th>0.0118</th>
<th>0.0128</th>
<th>0.0132</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0140</td>
<td>0.0150</td>
<td>0.0162</td>
<td>0.0173</td>
<td>0.0180</td>
<td>0.0200</td>
</tr>
<tr>
<td></td>
<td>0.0230</td>
<td>0.0250</td>
<td>0.0280</td>
<td>0.0320</td>
<td>0.0350</td>
<td>0.0410</td>
</tr>
<tr>
<td></td>
<td>0.0470</td>
<td>0.0540</td>
<td>0.0630</td>
<td>0.0720</td>
<td>0.0800</td>
<td>0.0920</td>
</tr>
<tr>
<td></td>
<td>0.1050</td>
<td>0.1200</td>
<td>0.1350</td>
<td>0.1480</td>
<td>0.1620</td>
<td>0.1770</td>
</tr>
<tr>
<td></td>
<td>0.1920</td>
<td>0.2070</td>
<td>0.2250</td>
<td>0.2440</td>
<td>0.2630</td>
<td>0.2830</td>
</tr>
<tr>
<td></td>
<td>0.3070</td>
<td>0.3310</td>
<td>0.3620</td>
<td>0.3940</td>
<td>0.4375</td>
<td>0.5000</td>
</tr>
</tbody>
</table>

The problem was discussed in (Sandgren 1990; Kannan and Kramer 1994) to demonstrate different algorithms for nonlinear mixed discrete-continuous optimization. The compromise DSP of this problem is as follows:

Given

- S = Allowable Stress 189,000 psi.
- G = Shear Modulus $1.15 \times 10^8$

Find

Deviation Function $d_1^+$

Satisfy

Constraints

\[ g_1, \text{ shear stress} \frac{8KF_{\max}D}{5\pi d^3} \leq 1.0 \]
\[ g_2, \text{ free length limit} \frac{l_{max}}{l_f} \geq 1.0 \]
\[ g_3, \text{ minimum wire diameter} \frac{d_{min}}{d} \leq 1.0 \]
\[ g_4, \text{ maximum outside diameter} \frac{D+d}{D_{max}} \leq 1.0 \]
\[ g_5, \text{ winding limit} \frac{C}{3} \leq 1.0 \]
\[ g_6, \text{ maximum preload deflection} \frac{\delta}{\delta_{pm}} \leq 0.0 \]
\[ g_7, \text{ combined deflection consistency} \left( \frac{\partial - F_{\max} - F_p}{K} - 1.5(N+2)d \right) \leq 1.0 \]
\[ g_8, \text{ deflection requirement} \frac{K\delta_{w}}{(F_{\max} - F_p)} \leq 1.0 \]

Goals

\[ F:\{0.25\pi^2Dd^2(N+2)\}/\text{Vol}_{TV} - d_1^+ = 0.0 \]

Bounds

\[ D_{LB} \cdot D \cdot D_{UB} \]
\[ d_{LB} \cdot d \cdot d_{UB} \]
\[ N_{LB} \cdot N \cdot N_{UB} \]

Minimize

Deviation Function $d_1^+$

F$_{\max}$ = Maximum Working Load 1000 lb.
I$_{max}$ = Maximum Free Length 14.00 in.
D$_{min}$ = Minimum Wire Diameter 0.200 in.
D$_{max}$ = Maximum Outside Spring Diameter 3.00 in.
F$_p$ = Preload Compression Force 300 lb.
$\delta_{pm}$ = Maximum Deflection under Preload 6.00 in.
$\delta_{w}$ = Deflection from Preload to Max Load 1.25

Spring is Guided

\[ C = D/d \]
\[ C_t = \frac{4C-1}{4C-4} + \frac{0.615}{C} \]
\[ K = \frac{8ND^2}{F_{\max}} \text{ lb / in.} \]
\[ \partial = \frac{F_{\max}}{K} \text{ in.} \]
\[ l_f = r + 1.05(N+2)d \]
\[ \partial_p = F_p/K \]
The bounds on the system variables for the problem are $D_{LB} = 1.0, D_{UB} = 6.0, d_{LB} = 0.0, d_{UB} = 0.5, N_{LB} = 3, N_{UB} = 30$. Based on the previous results on this problem, the target value for the cost goal is taken as $Vol_{TV} = 0.5$ in$^3$. In Table 2, the results from FALP, and (Sandgren 1990; Kannan and Kramer 1994) are compared. While Kannan's solution is 15.5% better than Sandgren's solution, the objective function (corresponding to the deviation function) found by FALP is 58.2% lower than the Sandgren's solution. The constraint values are all feasible.

<table>
<thead>
<tr>
<th>Variable</th>
<th>FALP</th>
<th>Kannan and Kramer</th>
<th>Sandgren</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$ (in.)</td>
<td>1.000</td>
<td>1.329</td>
<td>1.180</td>
</tr>
<tr>
<td>$d$ (in.)</td>
<td>0.283</td>
<td>0.283</td>
<td>0.283</td>
</tr>
<tr>
<td>$N$</td>
<td>3</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>$g_1$</td>
<td>0.874</td>
<td>1.054</td>
<td>0.971</td>
</tr>
<tr>
<td>$g_2$</td>
<td>0.129</td>
<td>0.318</td>
<td>0.382</td>
</tr>
<tr>
<td>$g_3$</td>
<td>0.707</td>
<td>0.707</td>
<td>0.707</td>
</tr>
<tr>
<td>$g_4$</td>
<td>0.428</td>
<td>0.537</td>
<td>0.488</td>
</tr>
<tr>
<td>$g_5$</td>
<td>0.849</td>
<td>0.639</td>
<td>0.720</td>
</tr>
<tr>
<td>$g_6$</td>
<td>0.016</td>
<td>0.089</td>
<td>0.089</td>
</tr>
<tr>
<td>$g_7$</td>
<td>-0.641</td>
<td>-0.200</td>
<td>-0.334</td>
</tr>
<tr>
<td>$g_8$</td>
<td>0.182</td>
<td>0.998</td>
<td>0.998</td>
</tr>
<tr>
<td>$F$ (in.$^3$)</td>
<td>0.988</td>
<td>2.365</td>
<td>2.799</td>
</tr>
</tbody>
</table>

Table 2. Coil Spring Results

3.2 Spring Design: Verification and Validation

Verification of Solution

In this section we verify the solution found by FALP and validate that it is the best possible solution for this problem. In this study we used 3 different starting points, the upper bounds of the variables, the lower bounds of the variables, and points in the middle of each range. Each starting point converged to the same solution. Since $D$ and $N$ are at their lower bounds in the solution we illustrate the variable activity using only the upper bounds of the variables, and points in the middle of each range. Before we discuss the solution, the notion of an "cycle" in FALP warrants some definitions. Since FALP consists of 2 solvers, the total number of cycles equals those spent in the foraging search and the ALP solver. A cycle in FALP is defined as the process to move from one design point to another. But because one is based on search heuristics and the other is based on calculus, the process of moving from one point to another is different. In the foraging search, a cycle is the search of one neighborhood and the selection of the next design point based on the foraging protocol. In this problem, the number of cycles (neighborhood searches) used was 100. In the ALP solver, a cycle is the linearization of the nonlinear model and solution of the linear model. The number of cycle in the ALP routine depends on the mathematics of the model. Convergence criteria is set for the continuous variables, and when this is reached, ALP stops. If convergence is not reached, there is an upper limit of 40 cycles in ALP.

For the spring problem, since the value of $D$, the continuous variable is at a lower bound, the continuous solver, ALP only is enacted for one cycle. The ALP solver keeps $D$ at the lower bound and concludes that convergence has been achieved. In Figures 6-8, the search history of the foraging algorithm for the three system variables. The one trait of the algorithm that is clearly illustrated is the diversification scheme. If a system variable remains at a given value for more than a specified number of cycles, the variable is changed to another value in completely different part of the design space. For instance, in Figure 6, the diversification scheme was enacted around cycles 36, 57, and 80. Similar behavior can be seen in Figures 7 and 8.

Figure 6. Coil Diameter Behavior

Figure 7. Number of Coils Behavior
Two of the fundamental observations of foraging animals we have modeled are schema identification and dynamic memory. We illustrate these two developments using Table 3 and Figure 9. In Table 3, the 10 best solutions for the spring problem as identified by the foraging portion of the algorithm are shown. Nine of the ten solutions include the value of 1.0 for D, while N and d vary. Since D is a continuous variable, it is not identified as being part of the schema. A designer can use this information to simplify the problem.

<table>
<thead>
<tr>
<th>Z</th>
<th>D</th>
<th>N</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.494</td>
<td>1</td>
<td>3</td>
<td>0.283</td>
</tr>
<tr>
<td>0.581</td>
<td>1</td>
<td>3</td>
<td>0.307</td>
</tr>
<tr>
<td>0.593</td>
<td>1</td>
<td>4</td>
<td>0.283</td>
</tr>
<tr>
<td>0.676</td>
<td>1</td>
<td>3</td>
<td>0.331</td>
</tr>
<tr>
<td>0.691</td>
<td>1</td>
<td>5</td>
<td>0.283</td>
</tr>
<tr>
<td>0.697</td>
<td>1</td>
<td>4</td>
<td>0.307</td>
</tr>
<tr>
<td>0.790</td>
<td>1</td>
<td>6</td>
<td>0.283</td>
</tr>
<tr>
<td>0.811</td>
<td>1</td>
<td>4</td>
<td>0.331</td>
</tr>
<tr>
<td>0.814</td>
<td>1</td>
<td>5</td>
<td>0.307</td>
</tr>
<tr>
<td>0.872</td>
<td>1.5</td>
<td>3</td>
<td>0.307</td>
</tr>
</tbody>
</table>

**Table 3. 10 Best Spring Solutions (Schema)**

In Figure 9, we validate the dynamic memory structure of the foraging algorithm. In Figure 9, the number of moves is plotted against the cycle number in the foraging algorithm for three different starting points. Again, a cycle is a complete neighborhood search. For two of the starting points, the number of moves not allowed increases rapidly up till about cycle 65, and then the algorithm becomes more and more content with the visited solutions. In other words, earlier in the solution process, a longer memory list is enacted. In a foraging context, the algorithm (or an animal in search of food) will not return to a site already visited for a considerable amount of time. As the solution process continues the memory list steadily decreases, as the algorithm (animal) becomes more satisfied with what has been found and will not try new directions as often.

The total number of function evaluations in this study is on the order of \( n^2 \times (m-1) \times l \) where \( n \) is the number of design variables, \( m \) is the number of neighbors allowed in the local neighborhood and \( l \) is the number of foraging cycles. The total number of possible functions evaluations (a measure of the problem size) is on the order of \( d_i \), where \( d_i \) is the number of discrete values for design variable number \( i \). For the spring study the number of function evaluations is

\[
3 \times 2 \times (4-1) \times 100 = 1800
\]

and the total possible is

\[
11 \times 27 \times 42 = 12474.
\]

Therefore, the percent of the discrete design space searched is approximately 14%. This is not a large percentage, but with the small size of the spring problem, even fewer foraging neighborhood searches could have been performed, as the best solution was encountered well before the search ended at cycles 10, 10, and 14 for the lower bound, middle point, and upper bound starting points. Optimizing the length of the search based on problem size is part of our future work in this area.

**Validation of Solution**

The success of our solution is interesting. As a means of verifying our solution, we performed an exhaustive combinatorial experiment using the discrete and integer variable. Since there are 28 possible values for N and 42 possible values for d, the total number of possible combinations is \( 28 \times 42 = 1176 \). At each of these combinations we performed a one-variable optimization problem with respect to the continuous variable D, keeping N and d constant. Therefore, we obtained 1176 different designs. Out of these 1176 designs, the solution found by FALP presented in Table 2 had the smallest objective function. The previous solutions from other studies were indeed found to be local optimum, but only local optimum. By performing this exhaustive search we are able to validate our solution and show that it is indeed a global solution for this problem.

**3.3 Pressure Vessel Design** (Sandgren 1990; Kannan and Kramer 1994; Hsu, Sun et al. 1995; Lin, Zhang et al. 1995)
This is a problem involving discrete and continuous variables. A cylindrical pressure vessel is capped at both ends by hemispherical heads as shown in Figure 10. The goal is to minimize the total cost of manufacturing the pressure vessel, including the cost of material, and cost of forming and welding. The design variables are R and L, the inner radius and length of the cylindrical section, and T_s and T_h, the thickness of the shell and head. The variables are each given in inches. The variables R and L are continuous, while T_s and head. The variables are each given in inches. The variables R and L are continuous, while T_s and T_h are integer multiples of 0.0625 inch, the available thicknesses of rolled steel plates. The constraints g_1, g_2, g_3 correspond to ASME limits on the geometry while g_4 corresponds to a minimum volume limit.

The problem was discussed in (Sandgren 1990; Kannan and Kramer 1994; Hsu, Sun et al. 1995; Lin, Zhang et al. 1995) to demonstrate different algorithms for nonlinear mixed discrete-continuous optimization. In the study by Lin, Genetic Algorithms (GA) and Simulated Annealing (SA) were used as the solution algorithms. The compromise DSP of this problem is as follows:

\[
\begin{align*}
\text{Goal} & : \text{Minimize} \quad F = \left[0.6224 T_s R + 1.7781 T_s R^2 + 3.1661 T_s^2 L + 19.84 T_s^2 R \right] / Cost_{TV} - d_1^+ = 0.0 \\
\text{Bounds} & : R_{LB} \cdot R \cdot R_{UB} \\
& \cdot T_s \cdot T_s \cdot T_s_{LB} \\
& \cdot T_h \cdot T_h \cdot T_h_{LB} \\
\text{Minimize} & : \text{Deviation Function } d_1^+ 
\end{align*}
\]

The bounds on the system variables for the problem are R_{LB} = 25 in., R_{UB} = 150 in., L_{LB} = 25 in., L_{UB} = 150 in., T_{sLB} = 0.0625 in., T_{sUB} = 1.25 in., T_{hLB} = 0.0625 in., T_{hUB} = 1.25 in.. Based on the previous studies of this problem, the target value for the cost goal is, Cost_{TV} = $5000.00. In this problem as in the spring problem, the number of foraging cycles (neighborhood searches) used was 100. There is an upper limit of 40 cycles in ALP, if convergence is not achieved.

In Table 4, the results from FALP are compared with the previous studies. The objective function (corresponding to the deviation function) found by FALP is 14.6% lower than the previous best found solution. The constraint values are all feasible (∗1).

### 3.4 Pressure Vessel Design: Verification and Validation

Again, in this problem we found favorable results. We used various starting points for this problem as in the spring design problem. We could have again performed an exhaustive search using 400 (20 discrete values for two variables) combinations of T_s and T_h. But each point would have required an optimization with respect to two variables. But by simply analyzing the active constraints and nature of the objective function, we can show where the room for improvement in the previous studies occurs. Specifically looking at the results from Hsu (Hsu, Sun et al. 1995), constraints g_1 (1.000) and g_2 (0.989) are active or very close to active. Since the cost is a monotonically increasing function with respect to every variable, the only improvement would be from decreasing a variable. But in g_1 and g_2, decreasing x_3, or x_4, will result in infeasibility. Therefore, any improvement must occur by...
changing \( x_1 \) or \( x_2 \). In constraint \( g_3 \) (0.424) we can change the value of \( x_2 \). But \( x_2 \) also occurs in \( g_4 \), therefore caution must be used as to not cause \( g_4 \) to become infeasible. But since \( g_4 \) is not active, there is some slack available. Keeping \( x_1 \) constant, we can decrease \( x_2 \) to 84.6 in. where \( g_4 \) becomes active. The new solution is shown in Table 5.

<table>
<thead>
<tr>
<th>Solution from Hsu</th>
<th>Improvement of Hsu's solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>R (in.)</td>
<td>51.81</td>
</tr>
<tr>
<td>L (in.)</td>
<td>101.85</td>
</tr>
<tr>
<td>( T_s ) (in.)</td>
<td>1.00</td>
</tr>
<tr>
<td>( T_h ) (in.)</td>
<td>0.50</td>
</tr>
<tr>
<td>( g_1 )</td>
<td>1.000</td>
</tr>
<tr>
<td>( g_2 )</td>
<td>0.989</td>
</tr>
<tr>
<td>( g_3 )</td>
<td>0.424</td>
</tr>
<tr>
<td>( g_4 )</td>
<td>0.831</td>
</tr>
<tr>
<td>( F ) ($)</td>
<td>7021.67</td>
</tr>
</tbody>
</table>

Table 5. Improvement in Hsu’s Solution (Hsu, Sun et al. 1995)

As is shown, a significant improvement in the cost of the pressure vessel is obtained simply by exploring the active constraints and monotonic behavior of the objective function. Similar logic can be used to show that the other studies (Sandgren 1990; Kannan and Kramer 1994) can be improved by altering one variable until a constraint becomes active. Although, we have not proven that the solution using foraging is the best global solution, we have shown dramatic improvement over the previous studies.

In Table 6, the 10 best solutions for the pressure vessel problem as identified by the foraging portion of the algorithm are shown. Nine of the ten solutions include the value of 37.5 for \( R \), and 250 for \( L \), while \( T_s \) and \( T_h \) vary. Since \( R \) and \( L \) are continuous variables, they are not identified as being part of the schema. A designer can use this information to simplify the problem.

<table>
<thead>
<tr>
<th>( Z )</th>
<th>( R )</th>
<th>( L )</th>
<th>( T_s )</th>
<th>( T_h )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.441</td>
<td>37.5</td>
<td>250</td>
<td>0.75</td>
<td>0.575</td>
</tr>
<tr>
<td>0.452</td>
<td>37.5</td>
<td>250</td>
<td>0.75</td>
<td>0.4675</td>
</tr>
<tr>
<td>0.464</td>
<td>37.5</td>
<td>250</td>
<td>0.75</td>
<td>0.5</td>
</tr>
<tr>
<td>0.475</td>
<td>37.5</td>
<td>250</td>
<td>0.75</td>
<td>0.5625</td>
</tr>
<tr>
<td>0.478</td>
<td>37.5</td>
<td>250</td>
<td>0.8125</td>
<td>0.375</td>
</tr>
<tr>
<td>0.484</td>
<td>50</td>
<td>115</td>
<td>0.9375</td>
<td>0.5</td>
</tr>
<tr>
<td>0.486</td>
<td>37.5</td>
<td>250</td>
<td>0.75</td>
<td>0.625</td>
</tr>
<tr>
<td>0.489</td>
<td>37.5</td>
<td>250</td>
<td>0.8125</td>
<td>0.375</td>
</tr>
<tr>
<td>0.497</td>
<td>37.5</td>
<td>250</td>
<td>0.75</td>
<td>0.6875</td>
</tr>
<tr>
<td>0.500</td>
<td>37.5</td>
<td>250</td>
<td>0.8125</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 6. 10 Best Pressure Vessel Solutions

In Figure 11, we again validate the dynamic memory structure of the foraging algorithm. In Figure 11, the number of moves is plotted against the cycle number in the foraging algorithm for three different starting points. For all three starting points, the number of moves not allowed increases, but in the later cycles begins to level off. With the lower bound starting point, the number of moves not allowed is significantly more than the other two starting points, indicating that the better regions are found more rapidly and therefore, a greater number of moves must be not allowed. This is supported by the fact that the best solution was found at cycle number 9 using the lower bound starting point.

![Figure 11. Number of Moves Not Allowed](image)

For the pressure vessel study the number of function evaluations is

\[ 4 \times 2 \times (4-1) \times 100 = 2400 \]

and the total possible is

\[ 11 \times 11 \times 20 \times 20 = 48400. \]

So, the percent of the discrete design space searched is approximately 5\%. This is even a smaller percentage than the spring problem because the size of this problem (number of variables and number of discrete values) is larger than the spring problem, but the same number of foraging cycles is used. Similar to the spring problem, even fewer foraging neighborhood searches could have been performed, as the best solution was encountered well before the search ended at cycles 9, 31, and 9 for the lower bound, middle point, and upper bound starting points. Optimizing the length of the search based on problem size is part of our future work in this area. This will include looking at the efficiency of the search process and determining if a certain search percentage of the design space produces superior solutions regardless of problem size.

4 CLOSURE

We present a new mixed discrete/continuous solution algorithm, namely, the Foraging-directed Adaptive Linear Programming Algorithm. The new discrete portion of the algorithm is based on the notion of foraging of animals in the wild and includes a dynamic memory structure and schema identification. The effectiveness of the solution approach presented is illustrated using two examples. These examples are strictly single objective, but have strictly been used as a means of comparison. A strength of the foraging algorithm for mixed discrete/continuous design problems is the ability to
handle multiple objectives. Although it has not been demonstrated in this paper, the algorithm uses a lexicographic routine to handle multiple objectives, which we feel are inherent in design problems. The FALP algorithm presented is the solution algorithm in the computer decision support package, Decision Support in the Design of Engineering Systems (DSIDES). Present work includes 1) enhancing the dynamic memory structure to more accurately model nature and to improve the efficiency of the search, and 2) defining appropriate neighborhood structure, modeling animal behavior as close as possible.

This work is a step towards a broader goal. By combining these aspects of the Tabu Search, Simulated Annealing, and Genetic Algorithms, we look to establish a broad class of solution algorithms, under which TS, GA, and SA can be classified. This broad class is an abstraction of nature, where intelligence is embedded in many biological processes. Algorithms such as GAs look to model a natural process and capture the inherent intelligence in a computational model. By using foraging as our foundation, we look to establish a class of algorithms based on intelligence, either innate or artificial. We find aspects of GAs, SA, and TS in the foraging analogy, and assert that there is a broad class of intelligent algorithms under which these can be grouped. The work in this paper was stimulated by the exploration of the similarities between optimization and artificial intelligence in (Glover 1986; Glover and Greenberg 1989).

ACKNOWLEDGMENTS

We gratefully acknowledge the following grants: NASA NGT 51102 and NASA NAG-1-1564. The cost of computer time was underwritten by the Systems Realization Laboratory of Georgia Institute of Technology.

REFERENCES


