A Template For Scatter Search And Path Relinking

Fred Glover

School of Business, CB 419
University of Colorado
Boulder, CO 80309-0419, USA

fred.glover@colorado.edu

Abstract. Scatter search and its generalized form called path relinking are evolutionary methods that have recently been shown to yield promising outcomes for solving combinatorial and nonlinear optimization problems. Based on formulations originally proposed in the 1960s for combining decision rules and problem constraints, these methods use strategies for combining solution vectors that have proved effective for scheduling, routing, financial product design, neural network training, optimizing simulation and a variety of other problem areas. These approaches can be implemented in multiple ways, and offer numerous alternatives for exploiting their basic ideas. We identify a template for scatter search and path relinking methods that provides a convenient and "user friendly" basis for their implementation. The overall design can be summarized by a small number of key steps, leading to versions of scatter search and path relinking that are fully specified upon providing a handful of subroutines. Illustrative forms of these subroutines are described that make it possible to create methods for a wide range of optimization problems. Highlights of these components include new *diversification generators* for zero-one and permutation problems (extended by a mapping-by-objective technique that handles additional classes of problems), together with processes to avoid generating or incorporating duplicate solutions at various stages (related to the avoidance of cycling in tabu search) and a new method for creating improved solutions.

*** UPDATED AND EXTENDED: February 1998 ***

Previous version appeared in *Lecture Notes in Computer Science, 1363*, J.K. Hao, E. Lutton, E. Ronald, M. Schoenauer, D. Snyers (Eds.), 13-54, 1997.

This research was supported in part by the Air Force Office of Scientific Research Grant #F49620-97-1-0271.

Table of Contents

1 Introduction

Scatter search and path relinking have recently been investigated in a number of studies, disclosing the promise of these methods for solving difficult problems in discrete and nonlinear optimization. Recent applications of these methods (and of selected component strategies within these methods) include:[1]

Vehicle Routing – Rochat and Taillard (1995); Taillard (1996)
Quadratic Assignment – Cung *et al*. (1996)
Financial Product Design – Consiglio and Zenios (1996)
Neural Network Training – Kelly, Rangaswamy and Xu (1996)
Job Shop Scheduling – Yamada and Nakano (1996)
Flow Shop Scheduling – Yamada and Reeves (1997)
Graph Drawing – Laguna and Marti (1997)
Linear Ordering – Laguna, Marti and Campos (1997)
Unconstrained Continuous Optimization – Fleurent *et al*. (1996)
Bit Representation – Rana and Whitley (1997)
Optimizing Simulation – Glover, Kelly and Laguna (1996)
Complex System Optimization – Laguna (1997)

We propose a template for generating a broad class of scatter search and path relinking methods, with the goal of creating versions of these approaches that are convenient to implement. Our design is straightforward, and can be readily adapted to optimization problems of diverse structures. We offer specific comments relating to multidimensional knapsack problems, graph partitioning problems, linear and nonlinear zero-one problems, mixed integer programming problems and permutation problems.

From the standpoint of classification, scatter search and path relinking may be viewed as evolutionary algorithms that construct solutions by combining others, and derive their foundations from strategies originally proposed for combining decision rules and constraints (Glover, 1963, 1965). The goal of these procedures is to enable a solution procedure based on the combined elements to yield better solutions than one based only on the original elements.

Historically, the antecedent strategies for combining decision rules were introduced in the context of scheduling methods, to obtain improved local decision rules for job shop scheduling problems, evaluated by simulations of consequences for makespan. New rules were generated by creating numerically weighted combinations of existing rules, suitably restructured so that their evaluations embodied a common metric. The approach was motivated by the supposition that information about the relative desirability of alternative choices is captured in different forms by different

[1] The References Section contains website listings where abstracts and/or copies can be obtained for a number of the references cited in this paper.

rules, and that this information can be exploited more effectively when integrated by means of a *combination mechanism* than when treated by the standard strategy of selecting different rules one at a time, in isolation from each other. In addition, the method departed from the customary approach of stopping upon reaching a local optimum, and instead continued to vary the parameters that determined the combined rules, as a basis for producing additional trial solutions. (This latter strategy also became a fundamental component of tabu search. See, e.g., Glover and Laguna, 1997.) The decision rules created from such combination strategies produced better empirical outcomes than standard applications of local decision rules, and also proved superior to a "probabilistic learning approach" that selected different rules probabilistcally at different junctures, but without the integration effect provided by generating combined rules (Crowston, *et al*., 1963).

The associated procedures for combining constraints likewise employed a mechanism of generating weighted combinations, in this case applied in the setting of integer and nonlinear programming, by introducing nonnegative weights to create new constraint inequalities, called *surrogate constraints*. The approach isolated subsets of constraints that were gauged to be most critical, relative to trial solutions based on the surrogate constraints, and produced new weights that reflected the degree to which the component constraints were satisfied or violated.

A principal function of surrogate constraints, in common with the approaches for combining decision rules, was to provide ways to evaluate choices that could be used to generate and modify trial solutions. From this foundation, a variety of heuristic processes evolved that made use of surrogate constraints and their evaluations. Accordingly, these processes led to the complementary strategy of combining solutions, as a *primal* counterpart to the *dual* strategy of combining constraints[2], which became manifest in scatter search and its path relinking generalization.

2 Foundations Of Scatter Search And Path Relinking

2.1 Scatter Search

The scatter search process, building on the principles that underlie the surrogate constraint design, is organized to (1) capture information not contained separately in the original vectors, (2) take advantage of auxiliary heuristic solution methods to evaluate the combinations produced and to generate new vectors.

The original form of scatter search (Glover, 1977) may be sketched as follows.

---

[2] Surrogate constraint methods give rise to a mathematical duality theory associated with their role as relaxation methods for optimization (e.g., see Greenberg and Pierskalla, 1970, 1973; Glover, 1965, 1975; Karwan and Rardin, 1976, 1979; Freville and Plateau, 1986, 1993).

Scatter Search Procedure

1. Generate a starting set of solution vectors by heuristic processes designed for the problem considered, and designate a subset of the best vectors to be *reference solutions*. (Subsequent iterations of this step, transferring from Step 3 below, incorporate advanced starting solutions and best solutions from previous history as candidates for the reference solutions.)
2. Create new points consisting of linear combinations of subsets of the current reference solutions. The linear combinations are:
   (a) chosen to produce points both inside and outside the convex regions spanned by the reference solutions.
   (b) modified by generalized rounding processes to yield integer values for integer-constrained vector components.
3. Extract a collection of the best solutions generated in Step 2 to be used as starting points for a new application of the heuristic processes of Step 1. Repeat these steps until reaching a specified iteration limit.

Three particular features of scatter search deserve mention. First, the linear combinations are structured according to the goal of generating weighted centers of selected subregions, allowing for nonconvex combinations that project these centers into regions external to the original reference solutions. The dispersion pattern created by such centers and their external projections is particularly useful for mixed integer optimization. (Appendix 1 gives specific procedures in this context.) Second, the strategies for selecting particular subsets of solutions to combine in Step 2 are designed to make use of clustering, which allows different types of strategic variation by generating new solutions "within clusters" and "across clusters". Third, the method is organized to use supporting heuristics that are able to start from infeasible solutions, and hence which remove the restriction that solutions selected as starting points for re-applying the heuristic processes must be feasible.[3]

In sum, scatter search is founded on the following premises.

(P1) Useful information about the form (or location) of optimal solutions is typically contained in a suitably diverse collection of elite solutions.

(P2) When solutions are combined as a strategy for exploiting such information, it is important to provide for combinations that can extrapolate beyond the regions spanned by the solutions considered, and further to incorporate heuristic processes to map combined solutions into new points. (This serves to provide both diversity and quality.)

---

[3] An incidental feature – that has more than incidental implications – is the incorporation of general (mixed integer) solution vectors, in contrast to a reliance on binary representations. Although methods incorporating non-binary vectors have long existed in domains outside of those that operate by combining solution vectors, the GA proposals remained wedded to binary representations until the mid to late 1980s. As shown in Glover (1994a), a reliance on such representations can create "information gaps" for combining solutions. The problems of *distortion* in binary-based GAs are therefore not surprising.

(P3) Taking account of multiple solutions simultaneously, as a foundation for creating combinations, enhances the opportunity to exploit information contained in the union of elite solutions.

The fact that the heuristic processes of scatter search are not restricted to a single uniform design, but represent a varied collection of procedures, affords additional strategic possibilities.

2.2 Path Relinking

From a spatial orientation, the process of generating linear combinations of a set of reference solutions may be characterized as generating paths between and beyond these solutions, where solutions on such paths also serve as sources for generating additional paths. This leads to a broader conception of the meaning of creating *combinations* of solutions. By natural extension, such combinations may be conceived to arise by generating paths between and beyond selected solutions in neighborhood space, rather than in Euclidean space (Glover 1989, 1994b).

This conception is reinforced by the fact that a path between solutions in a neighborhood space will generally yield new solutions that share a significant subset of attributes contained in the parent solutions, in varying "mixes" according to the path selected and the location on the path that determines the solution currently considered. The character of such paths is easily specified by reference to solution attributes that are added, dropped or otherwise modified by the moves executed in neighborhood space. Examples of such attributes include edges and nodes of a graph, sequence positions in a schedule, vectors contained in linear programming basic solutions, and values of variables and functions of variables. To generate the desired paths, it is only necessary to select moves that perform the following role: upon starting from an *initiating solution*, the moves must progressively introduce attributes contributed by a *guiding solution* (or reduce the distance between attributes of the initiating and guiding solutions). The process invites variation by interchanging the roles of the initiating and guiding solutions, and also by inducing each to move simultaneously toward the other as a way of generating combinations.[4]

Such an incorporation of attributes from elite parents in partially or fully constructed solutions was foreshadowed by another aspect of scatter search, embodied in an accompanying proposal to assign preferred values to subsets of *consistent* and *strongly determined* variables. The theme is to isolate assignments that frequently or influentially occur in high quality solutions, and then to introduce compatible subsets of these assignments into other solutions that are generated or amended by heuristic procedures. (Such a process implicitly relies on a simple form of frequency based memory to identify and exploit variables that qualify as

[4] Variants of path relinking that use constructive and destructive neighborhoods, called *vocabulary building* approaches, produce strategic combinations of partial solutions (or "solution fragments") as well as of complete solutions. The organization of vocabulary building permits the goal for combining the solution components to be expressed as an optimization model in a number of contexts, with the added advantage of allowing exact methods to be used to generate the moves (see, e.g., Glover and Laguna, 1997).

consistent, and thereby provides a bridge to associated tabu search ideas discussed in later sections.)

Multiparent path generation possibilities emerge in path relinking by considering the combined attributes provided by a set of guiding solutions, where these attributes are weighted to determine which moves are given higher priority. The generation of such paths in neighborhood space characteristically "relinks" previous points in ways not achieved in the previous search history, hence giving the approach its name.

Neighborhoods for these processes may differ from those used in other phases of search. For example, they may be chosen to *tunnel through* infeasible regions that may be avoided by other neighborhoods. Such possibilities arise because feasible guiding points can be coordinated to assure that the process will re-enter the feasible region, with out danger of becoming "lost." The ability of neighborhood structures to capture contextual features additionally provides a foundation for incorporating domain-specific knowledge about different classes of problems, thus enabling path relinking to exploit such knowledge directly.[5]

2.3 Associated Considerations

The exploitation of strongly determined and consistent variables, as alluded to in the preceding section, is particularly important in scatter search and path relinking. The identities of such variables depend on the subsets of elite solutions chosen for defining them, and in the present setting these subsets of solutions are the ones made up of reference solutions.

Consistent variables, which receive a particular value (or small range of values) in a significant proportion of the solutions of the chosen subset, may be restricted to those that receive such a value in all solutions if the subset considered is relatively small (e.g., containing up to 3 or 4 solutions). It is likely that the identities of such variables may vary substantially over different subsets. The proportion of solutions used to define consistency may, however, be modified depending both on the size of the subject and the type of problem considered.

Strongly determined variables, which cause significant deterioration in the quality or feasibility of one or more solutions of the subset – if their value is changed in the indicated solution(s) – are similarly likely to have somewhat different identities in different subsets. Strongly determined variables do not have to qualify as consistent. Alternatively, they may be consistent over some set of solutions different from the one under consideration. In many 0-1 formulations, it is appropriate to focus on variables that receive a value of 1 as those that may qualify as strongly determined, as

---

[5] This may be contrasted with the GA crossover concept, which lacks a unifying design to handle applications that are not well served by the original proposals for exchanging components of binary vectors. The crossover model, which has significantly influenced the popular view of the meaning of combining vectors, compels one of two outcomes: (a) domain-specific knowledge must be disregarded (a property of GAs once regarded to be a virtue); (b) amended versions of crossover must repeatedly be devised, without a guiding principle for structuring the effort to accommodate new problems. The second outcome has led to frequent reliance on ad hoc constructions, as noted by Reeves (1997) and Muhlenbein (1997).

in "multiple choice" models where various subsets of variables must sum to 1. (In such formulations a variable would be classed as strongly determined relative to a particular solution if changing its value from 1 to 0 significantly damages the solution, without allowing a trivial fix.)

A customary approach for handling strongly determined and consistent variables, as proposed in the paper that introduced scatter search, is to temporarily confine them to their preferred values or ranges, while other variables are allowed to be freely manipulated. Once no further improvement is possible (by the rules of the method employed, or within a specified limit of computation) the confining conditions are relaxed to seek a final stage of improvement. Constructive heuristics can often be useful for initially assigning values to variables other than those in the restricted sets, and hence the outcome of evaluating and extending the specified sets of variables may be viewed as building solution combinations from solution fragments (composed of the value assignments to the restricted variables only), rather than from complete solutions. This in general is the theme of the vocabulary building procedure, which is a fundamental variant of scatter search and path relinking.[6]

Two natural approaches exist for determining values to be given to variables that are not among those selected to be restricted.

Approach 1. Build the combined solution by starting from the partial solution that assigns each consistent variable the value it receives (uniformly or most often) in the currently chosen set of reference solutions.

Approach 2. Build the combined solution by starting from a partial solution based on assigning values to some number of variables that are strongly determined but not consistent. (These strongly determined variables may be screened by a measure of consistency applied to a set of solutions different from the current one.)

In Approach 2, once a partial solution is generated by assigning values to a small number of strongly determined variables, then the values of other variables can be generated. This may require a more judicious process than starting from a partial solution created from consistent variables as in Approach 1, since the value to be assigned to a consistent variable is already predetermined to be the value that caused it to be defined consistent. However, by considering strongly determined variables that are not consistent, it is necessary to choose assignments for variables that do not invariably (or with high frequency) receive the same value in the set under consideration.[7]

A constructive process can create an undesirable mix of value assignments unless each step of assigning a value is closely monitored to determine the effect of the assignment on the solution being constructed. For a variety of problems, effective forms of constructive and destructive methods can be based on evaluations derived from surrogate constraint "ratio" evaluations (Glover 1965, 1977; Freville and

---

[6] The scatter search strategy of building solutions from fragments of others, as a supplementary mechanism for combining solutions, is refined in the vocabulary buildilng process by iteratively assembling and disassembling the fragments, with the ultimate goal of creating and assembling fragments that produce elite solutions.

[7] A Min K-Tree example from Glover and Laguna (1997) illustrates the merit of using a threshold evaluation to isolate high quality solution components.

Plateau, 1986). In the process of creating the partial solution, and especially as additional variables are subsequently selected and assigned values (after the partial solution is completed), an appropriate value to assign a given variable may differ from any value that the variable received in the set of reference solutions. This applies to consistent variables as well — i.e., a variable that received a particular value in all solutions of the reference set may no longer appropriately receive this value after a partial solution is constructed by Approach 2. (Evidently, the observation also holds for a partial solution constructed by Approach 1, if a consistent variable is one that receives a particular value in a certain fraction, but not all, of the reference solutions.) An approach that temporarily restricts possible assignments to those found in the union of some larger set of elite solutions is a common companion strategy to one that temporarily restricts values of selected strongly determined and consistent variables. Such considerations suggest that the reference solutions of scatter search and path relinking should be called "donors" rather than "parents." That is, they "donate" certain parts of themselves to provide a foundation for creating new solutions. The associated approach of generating *structured combinations* (Glover, 1994b) extends these notions by transforming the donor solutions into *rules* for generating solutions.

Building solutions from common values, as in Approach 1, gives a good approach for tightly constrained problems such as those arising in scheduling and partitioning applications, where constructive processes are important for achieving feasibility. An initial partial solution created from all (or any subset) of the variables that receive a common value is assured to have feasible completions.

2.4 Avoiding Duplication Constructions.

An important aspect of carrying out these processes is to avoid re-constructing solutions already generated. The avoidance of duplications by controlling the combined solutions, which includes submitting them to constructive and improving heuristics, can be a significant factor in producing an effective overall procedure. Such an avoidance can be viewed as a precondition for creating an appropriate level of diversification. Accordingly, this is a point where adaptive memory, as introduced in tabu search, becomes relevant.

A useful type of memory in the present context is provided by a "critical event" design, where variable numbers of steps during a constructive solution process are governed by choices that penalize assignments which tend toward solutions previously generated. In the 0-1 setting, for example, a frequency vector can be created by summing the critical solutions (those selected to be avoided). Frequency-based control can then be applied by penalizing a value assignment of 1 or 0 according to whether a particular variable receives a value in the summed vector that is larger or smaller than an associated threshold (which may differ by variable and the value to be penalized). This approach has proved effective in tabu search approaches for multidimensional knapsack and 0-1 quadratic programming problems (Glover and Kochenberger, 1996, 1997, and Hanafi and Freville, 1997).

Interesting variations exist for creating a memory scheme that will propel the approach toward new solutions without too strongly restricting the choices. For example, a conditional memory can be used, where the penalties or inducements change at each step by removing consideration of solutions in the critical set that evidently cannot be generated, given the values assigned so far. The elimination of such donor solutions then accordingly removes whatever influence they might have on penalizing future assignments of values to variables.

3 Outline Of The Scatter Search/Path Relinking Template

Drawing on the observations of the preceding section, we now describe a template for implementing both scatter search and path relinking that takes advantage of their common characteristics and principles. Components of this template consist of specific subroutines of the following types:

(1) A Diversification Generator: to generate a collection of diverse trial solutions, using an arbitrary trial solution (or seed solution) as an input.

(2) An Improvement Method: to transform a trial solution into one or more enhanced trial solutions. (Neither the input nor output solutions are required to be feasible, though the output solutions will more usually be expected to be so. If no improvement of the input trial solution results, the "enhanced" solution is considered to be the same as the input solution.)

(3) A Reference Set Update Method: to build and maintain a Reference Set consisting of the b best solutions found (where the value of b is typically small, e.g., between 20 and 40), organized to provide efficient accessing by other parts of the method.

(4) A Subset Generation Method: to operate on the Reference Set, to produce a subset of its solutions as a basis for creating combined solutions.

(5) A Solution Combination Method: to transform a given subset of solutions produced by the Subset Generation Method into one or more combined solution vectors.

We provide illustrative designs for each of these subroutines except the last, which will typically vary according to the context. The processes for combining solutions in scatter search and path relinking, as embodied in (5), have been alluded to in previous sections and are discussed at greater length in the references cited in the Introduction, especially in Glover and Laguna (1997). Consequently, we focus instead on other features, which deserve greater attention than they have been accorded in the past. Our purpose in particular is to identify forms of the subroutines (1) to (4), preceding, that can be usefully adapted to a variety of settings, and that can be integrated to produce an effective overall method.

We specify the general template in outline form as follows. This template reflects the type of design often used in scatter search and path relinking.[8]

---

[8] Other alternatives for organizing and implementing such methods can be found in Glover (1994a, 1995).

SS/PR Template

Initial Phase

1. *(Seed Solution Step.)* Create one or more seed solutions, which are arbitrary trial solutions used to initiate the remainder of the method.
2. *(Diversification Generator.)* Use the Diversification Generator to generate diverse trial solutions from the seed solution(s).
3. *(Improvement and Reference Set Update Methods.)* For each trial solution produced in Step 2, use the Improvement Method to create one or more enhanced trial solutions. During successive applications of this step, maintain and update a Reference Set consisting of the b best solutions found.
4. *(Repeat.)* Execute Steps 2 and 3 until producing some designated total number of enhanced trial solutions as a source of candidates for the Reference Set.

Scatter Search/Path Relinking Phase

5. *(Subset Generation Method.)* Generate subsets of the Reference Set as a basis for creating combined solutions.
6. *(Solution Combination Method.)* For each subset X produced in Step 5, use the Solution Combination Method to produce a set C(X) that consists of one or more combined solutions. Treat each member of C(X) as a trial solution for the following step.
7. *(Improvement and Reference Set Update Methods.)* For each trial solution produced in Step 6, use the Improvement Method to create one or more enhanced trial solutions, while continuing to maintain and update the Reference Set.
8. *(Repeat.)* Execute Steps 5-7 in repeated sequence, until reaching a specified cutoff limit on the total number of iterations.

Table 1, following, summarizes the relationships between sources of input and output solutions in the preceding template.

Source of Input Solutions	**Source of Output Solutions**
Arbitrary seed solutions	Diversification Generator
Diversification Generator	Improvement Method
Improvement Method	Reference Set Update Method
Reference Set Update Method	Subset Generation Method
Subset Generation Method	Solution Combination Method
Solution Combination Method	Improvement Method

Table 1: Input/Output Links

Scatter search and path relinking are often implemented in connection with tabu search (TS), and their underlying ideas share a significant intersection with the TS perspective. A principal element of this perspective is its emphasis on establishing a strategic interplay between intensification and diversification. In the original scatter

search design, which carries over to the present template, intensification is achieved by:

(a) the repeated use of the Improvement Method as a basis for refining the solutions created (from combinations of others);
(b) maintaining the Reference Set to consist of the highest quality solutions found; and
(c) choosing subsets of the Reference Set and uniting their members by strategies that reinforce the goal of generating good solutions (as opposed to relying on mating and "crossover" schemes that are heavily based on randomization).

In the mid to late 1980s, a number of the elements proposed earlier in scatter search began to be introduced in hybrid variants of GA procedures. Consequently, some of the current descendants of these hybrid approaches appear to have a structure similar to the outline of the SS/PR Template. Nevertheless, significant differences remain, due to perspectives underlying scatter search and path relinking that have not become incorporated into the GA hybrids. These are particularly reflected in the concepts underlying intensification and diversification, which will be elaborated in subsequent discussions.

The remaining sections are devoted to providing illustrative forms of the subroutines that support the foregoing template. We first focus on the diversification generator, followed by the method for updating the reference set and then the method for choosing subsets of the reference solutions. Finally we examine the issue of specifying an improvement method, and identify an approach that likewise embodies ideas that have not been adequately considered in the literature.

4 Diversification Generator

We indicate two simple types of diversification generators, one for problems that can be formulated in a natural manner as optimizing a function of zero-one variables, and the other for problems that can more appropriately be formulated as optimizing a permutation of elements. In each of these instances we disregard the possible existence of complicating constraints, in order to provide a simple representation of the basic ideas. However, these methods can also be used in the presence of such constraints by using the solutions generated as targets for creating solutions that satisfy the additional requirements of feasibility. This can be done by applying neighborhood procedures (including those that use constructive or destructive neighborhoods) to insure the preservation or attainment of feasibility, while utilizing evaluations that give preference to moves which approach the targets. In addition, Appendix 1 shows how zero-one solution generators can be embedded in a method to create diversified collections of feasible points for mixed integer programming and nonlinear optimization problems.

These approaches embody the tabu search precept that diversification is not the same as randomization. In this respect, they differ from the randomized approaches for creating variation that are typically proposed in other types of evolutionary

approaches. The goal of diversification is to produce solutions that differ from each other in significant ways, and that yield productive (or "interesting") alternatives in the context of the problem considered. By contrast, the goal of randomization is to produce solutions that may differ from each other in any way (or to any degree) at all, as long as the differences are entirely "unsystematic". From the tabu search viewpoint, a reliance on variation that is strategically generated can offer advantages over a reliance on variation that is distinguished only by its unpredictability.

4.1 Diversification Generators for Zero-One Vectors

We let x denote an n-vector each of whose components x_j receives the value 0 or 1. The first type of diversification generator we consider takes such a vector x as its seed solution, and generates a collection of solutions associated with an integer h = 1, 2,..., h*, where h* ≤ n - 1. (Recommended is h* ≤ n/5.)

We generate two types of solutions, x' and x'', for each value of h, by the following rule:

Type 1 Solution: Let the first component x'_1 of x' be 1 - x_1, and let x'_{1+kh} = 1 - x_{1+kh} for k = 1, 2, 3,..., k*, where k* is the largest integer satisfying k* ≤ n/h. Remaining components of x' equal 0.

To illustrate for x = (0,0,...,0): The values h = 1, 2 and 3 respectively yield x' = (1,1,...,1), x' = (1,0,1,0,1 ...) and x' = (1,0,0,1,0,0,1,0,0,1,....). This progression suggests the reason for preferring h* ≤ n/5. As h becomes larger, the solutions x' for two adjacent values of h differ from each other proportionately less than when h is smaller. An option to exploit this is to allow h to increase by an increasing increment for larger values of h.

Type 2 Solution: Let x'' be the complement of x'.

Again to illustrate for x = (0,0,...,0): the values h = 1, 2 and 3 respectively yield x'' = (0,0,...,0), x'' = (0,1,0,1,....) and x'' = (0,1,1,0,1,1,0,...). Since x'' duplicates x for h = 1, the value h = 1 can be skipped when generating x''.

We extend the preceding design to generate additional solutions as follows. For values of h ≥ 3 the solution vector is shifted so that the index 1 is instead represented as a variable index q, which can take the values 1, 2, 3, ..., h. Continuing the illustration for x = (0,0,...,0), suppose h = 3. Then, in addition to x' = (1,0,0,1,0,0,1,...), the method also generates the solutions given by x' = (0,1,0,0,1,0,0,1,...) and x' = (0,0,1,0,0,1,0,0,1....), as q takes the values 2 and 3.

The following pseudo-code indicates how the resulting diversification generator can be structured, where the parameter MaxSolutions indicates the maximum number

of solutions desired to be generated. Comments within the code appear in italics, enclosed within parentheses.

First Diversification Generator for Zero-One Solutions.

NumSolutions = 0
For h = 1 to h*
  Let q* = 1 if h < 3, and otherwise let q* = h
    (q denotes the value such that q will range from 1 to q*. We set q* = 1 instead of q* = h for h < 3 because otherwise the solutions produced for the special case of h < 3 will duplicate other solutions or their complements.)*
  For q = 1 to q*
    let k* = (n-q)/h <rounded down>
    For k = 1 to k*
      $x'_{q+kh} = 1 - x_{q+kh}$
    End k
    If h > 1, generate x'' as the complement of x'
      (x' and x'' are the current output solutions.)
    NumSolutions = NumSolutions + 2 (or + 1 if h = 1)
    If NumSolutions ≥ MaxSolutions, then stop generating solutions.
  End q
End h

The number of solutions x' and x'' produced by the preceding generator is approximately q*(q*+1). Thus if n = 50 and h* = n/5 = 10, the method will generate about 110 different output solutions, while if n = 100 and h* = n/5 = 20, the method will generate about 420 different output solutions.

Since the number of output solutions grows fairly rapidly as n increases, this number can be limited, while creating a relatively diverse subset of solutions, by allowing q to skip over various values between 1 and q*. The greater the number of values skipped, the less "similar" the successive solutions (for a given h) will be. Also, as previously noted, h itself can be incremented by a value that differs from 1.

For added variation:

If further variety is sought, the preceding approach can be augmented as follows. Let h = 3,4,..., h*, for h ≤ n - 2 (preferably h* ≤ n/3). Then for each value of h, generate the following solutions.

Type 1A Solution: Let x'_1 = 1 - x_1 and x'_2 = 1 - x_2. Thereafter, let x'_{1+kh} = 1 - x_{1+kh} and let x'_{2+kh} = 1 - x_{2+kh}, for k = 1,2,...,k*, where k* is the largest integer such that 2 + kp ≤ n. All other components of x' are the same as in x.

Type 2A Solution: Create x'' as the complement of x', as before.

Related variants are evident. The index 1 can also be shifted (using a parameter q) in a manner similar to that indicated for solutions of type 1 and 2.

4.2 A Sequential Diversification Generator

The concept of diversification invites a distinction between solutions that differ from a given solution (e.g., a seed solution) and those that differ from each other.[9] Our preceding comments refer chiefly to the second type of diversification, by their concern with creating a collection of solutions whose members exhibit certain contrasting features.

Diversification of the first type can be emphasized in the foregoing design by restricting attention to the complemented solutions denoted by x'' when h becomes larger than 2. In general, diversification of the second type is supported by complementing larger numbers of variables in the seed solution. We stress that this type of diversification by itself is incomplete, and the relevance of diversification of the first type is important to heed in many situations.

Approaches that combine characteristics of both types of diversification can be founded on the ideas of *sequential diversification* (Glover and Laguna, 1997). A "diverse sequence" can be composed by a rule that selects each successive vector to maximize the minimum distance from all vectors previously considered, according to a chosen distance metric. (There can sometimes be many vectors that qualify to be chosen as the next member of a partially constructed sequence by such a criterion. Ties can be broken by a perturbed distance function where distances to vectors that appear later in the partial sequence are considered smaller than distances to vectors that appear earlier in the sequence.) Sequential diversification makes it possible to generate solutions that differ from a *set* of solutions (such as a set of solutions examined during a search process), rather than those that differ from a single solution, by considering the members of the set to be first members of the sequence.

A sequential diversification generator for 0-1 vectors that follows the prescription to maximize the minimum distance from preceding vectors is embodied in the following procedure. We say that a solution y complements x over an index set J if $y_j = 1 - x_j$ for $j \in J$ and $y_j = x_j$ for $j \notin J$.

Sequential (Max/Min) Diversification Generator

1. Designate the seed solution x and its complement to be the first two solutions generated.
2. Partition the index set $N = \{1,\ldots,n\}$ for x into two sets N' and N'' that, as nearly as possible, contain equal numbers of indexes. Create the two

[9] These distinctions have often been overlooked by the genetic algorithm community, in spite of the emergence of GA hybrids that take advantage of tabu search to obtain improved outcomes. The interplay between intensification and diversification is sometimes further obscured by confusing it with the control theory notion of "exploitation versus exploration," which GAs have traditionally adopted. The nature and consequences of these differing ideas are discussed, for example, in Glover and Laguna (1997).

solutions x' and x'' so that x' complements x over N' and x'' complements x over N''.

3. Define each subset of N that is created by the most recent partition of N to be a key subset. If no key subset of N contains more than 1 element, stop. Otherwise partition each key subset S of N into two sets S' and S'' that contain, as nearly as possible, equal numbers of elements. (For the special case where S may contain only 1 element, designate one of S' and S'' to be the same as S, and the other to be empty.) Overall, choose the designations S' and S'' so that the number of partitions with |S'| > |S''| equals the number with |S''| > |S'|, as nearly as possible.
4. Let N' be the union of all subsets S' and let N'' be the union of all subsets S''. Create the complementary solutions x' and x'' relative to N' and N'' as in Step 2, and then return to Step 3. (The partition of each critical set into two parts in the preceding execution of Step 3 will cause the number of critical sets in the next execution of Step 3 to double.)

The foregoing process generates approximately *2(1 + log n)* solutions. If n is a power of 2, every solution produced maximizes the minimum Hamming distance from all previous solutions generated. (This maxmin distance, measured as the number of elements by which the solutions differ, is n/2 for every iteration after the first two solutions are generated in Step 1. Such a maxmin value is also approximately achieved when n is not a power of 2.)

In particular, starting with k = n, and updating k at the beginning of Step 3 by setting k: = k/2 (rounding fractional values upward), the number of elements in each key subset is either k or k-1. Thus, the method stops when k = 1. The balance between the numbers of sets S' and S'' of different sizes can be achieved simply by alternating, each time a set S with an odd number of elements is encountered, in specifying the larger of the two members of the partition to be S' or S''.

Useful variations result by partitioning N in different ways, but different partitions do not always produce different results. For example, if the initial approach uses a convenient *ascending index rule* that partitions each set so that the first member of the partition only contains indexes that are smaller than the minimum index contained in the second, then a modified basis for this rule that reindexes the last fourth of the elements of N to become instead the second fourth of these elements will produce exactly the same set of solutions as produced by the original indexing. A simple way to create additional partitions that does not have the indicated weakness, but that still takes advantage of the ascending index rule, is as follows.

For each of several selected prime numbers p, generate the sequence jp (modulo n) as the index j runs from 1 to n. The value 0 of the sequence is replaced by the value n. (E.g., if n = 8 and p = 3, the sequence is 3, 6, 1, 4, 7, 2, 5, 8.) The variables are then reindexed so that, for j = 1, 2, ..., n, the current index jp becomes the new index j. (In the special case where p divides n, the sequence cycles. Such divisors of n can be omitted, or else the sequence can be shifted to become k + jp (modulo n), where j = 1 to n/p and, in an outer loop, k ranges from 0 to n/p - 1.)

Preferably, only a relatively small number of prime numbers should be chosen to generate such different partitions, since the Max/Min Generator complements approximately half of the variables in each solution produced, and the proportion of variables complemented should reasonably be varied. This type of variation can be achieved indirectly, but effectively, by altering the seed solution for the generator. A natural approach is to allow seed solutions to be given by the best solutions found (e.g., a limited number of members of the Reference Set, which are selected to be somewhat different from each other). This generates solutions that are diverse in relation to those created by other problem solving processes employed.

The Max/Min Generator may also be usefully coupled with the first diversification generator to produce a composite approach. This provides a controlled means for varying the number of variables that are complemented. For instance, the Max/Min Generator (including the variant for reindexing relative to selected prime numbers) can begin with each of the following seed solutions produced by the earlier diversification generator.

(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...)

(1, 0, 0, 1, 0, 0, 1, 0, 0, 1, ...)
(0, 1, 0, 0, 1, 0, 0, 1, 0, 0, ...)
(0, 0, 1, 0, 0, 1, 0, 0, 1, 0, ...)

(1, 0, 0, 0, 1, 0, 0, 0, 1, 0, ...)
(0, 1, 0, 0, 0, 1, 0, 0, 0, 1, ...)
(0, 0, 1, 0, 0, 0, 1, 0, 0, 0, ...)
(0, 0, 0, 1, 0, 0, 0, 1, 0, 0, ...)

and so on.

Complemented solutions are excluded in the preceding representation, since these are automatically created by the Max/Min Generator. The solution (1, 0, 1, 0, 1, 0, ...) is also excluded, because it is likewise among those produced by the Max/Min Generator from the initial (0, 0, 0, 0, 0, 0, ...) solution, using the ascending index rule.

The foregoing 0-1 Diversification Generator and the one first described can additionally be applied to generate solutions that differ from a set of solutions by using a TS frequency-based approach. An example occurs by weighting the vectors of the set according to their importance, and creating a combined vector of weighted frequencies by summing the weighted vectors. The resulting vector can then be transformed into a seed solution for the Diversification Generator by mapping each of its components into 0 or 1, according to whether the value of the component lies below or above a chosen threshold.

The next section offers another type of organization to assure that new solutions differ from the seed solution as well as from each other.

4.3 Diversification Generator for Permutation Problems

Although permutation problems can be formulated as 0-1 problems, they constitute a special class that preferably should be treated somewhat differently. Assume that a given trial permutation P used as a seed is represented by indexing its elements so they appear in consecutive order, to yield P = (1,2, ..., n). Define the subsequence P(h:s), where s is a positive integer between 1 and h, to be given by P(h:s) = (s, s+h, s+2h, ..., s+rh), where r is the largest nonnegative integer such that s+rh ≤n. Then define the permutation P(h), for h ≤ n, to be P(h) = (P(h:h), P(h:h-1), ..., P(h:1)).

Illustration:

Suppose P is given by

P = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18)

If we choose h = 5, then P(5:5) = (5,10,15), P(5:4) = (4,9,14), P(5:3) = (3,8,13,18), P(5:2) = (2,7,12,17), P(5:1) = (1,6,11,16), to give:

P(5) = (5, 10, 15, 4, 9, 14, 3, 8, 13, 18, 2, 7, 12, 17, 1, 6, 11, 16)

Similarly, if we choose h = 4 then P(4:4) = (4,8,12,16), P(4:3) = (3,7,11,15), P(4:2) = (2,6,10,14,18), P(4:1) = (1,5,9,13,17) to give:

P(4) = (4, 8, 12, 16, 3, 7, 11, 15, 2, 6, 10, 14, 18, 1, 5, 9, 13, 17)

In this illustration we have allowed h to take the two values closest to the square root of n. These values are interesting based on the fact that, when h equals the square root of n, the minimum relative separation of each element from each other element in the new permutation is maximum, compared to the relative separation of exactly 1 in the permutation P. In addition, other useful types of separation result, and become more pronounced for larger values of n.

In general, for the goal of generating a diverse set of permutations, preferable values for h range from 1 to n/2. We also generate the reverse of the preceding permutations, denoted by P*(h), which we consider to be more interesting than P(h). The preference of P*(h) to P(h) is greater for smaller values of h. For example, when h = 1, P(h) = P and P*(h) is the reverse of P. (Also, P(n) = P*(1).) In sum, we propose a Diversification Generator for permutation problems to be one that generates a subset of the collection P(h) and P*(h), for h = 1 to n/2 (excluding P(1) = P).

4.4 Additional Role for the Diversification Generator

A diversification generator that takes one of the forms indicated above can be used to modify the balance between intensification and diversification in the SS/PR Template. In particular, a stronger diversification emphasis can be effected by means of optional additions to Steps 6 and 7, which we denote by 6A and 7A, as follows.

6A. *(Pre-Improvement Diversification.)* Use selected trial solutions produced in Step 6 as seeds to produce additional trial solutions by the Diversification Generator.

7A. *(Post-Improvement Diversification.)* Similarly, use selected trial solutions produced in Step 7 to produce additional trial solutions by the Diversification Generator, where each resulting trial solution in this case is immediately subjected again to Step 7 to produce an enhanced solution.

The provision to apply Step 7 again in the Post-Improvement Diversification is not required in the Pre-Improvement Diversification because Step 6 (and hence 6A) is directly followed by Step 7 in the SS/PR Template. Typically the optional steps 6A and 7A will be used to generate a single additional trial solution for each trial solution selected to be a seed solution. The Diversification Generator of the next section can easily be restricted to generate such a solution that is "far from" the seed solution by reference to the observations previously indicated. The potential value of these optional steps for a given application can of course be conveniently determined by implementing a version of the SS/PR Template that excludes them, and then comparing the outcomes with those produced by incorporating Step 6A and/or Step 7A.

5 Maintaining And Updating The Reference Set

The Reference Set Update method accompanies each application of the Improvement Method, and we examine this updating function first because of its linking role, which also introduces structures that provide a foundation for the Subset Generation Method. The update operation consists of maintaining a record of the b best solutions found, where value of b is treated as a chosen constant, but may readily be allowed to vary. The underlying issues are conceptually straightforward, but their ramifications within the SS/PR Template are sufficiently extensive to motivate a concrete design for such a procedure.[10]

Hereafter, we will refer to the Reference Set by the abbreviation "RefSet". We provide a pseudo-code description of the procedure for maintaining RefSet, which we call the *RefSet Update Routine*, that is organized to handle vectors of 0-1 variables. Related forms to handle permutation vectors follow the same general format.

5.1 Notation and Initialization

Let bNow denote the current number of solutions in the Reference Set. bNow begins at 0 in an Initialization Step, and is increased each time a new solution is added to the Reference Set, until reaching the value bMax (as where bMax may take a value from 10 to 40, for example).

At each step, RefSet stores the best solutions in an array x[i], i = 1 to bNow. The solution vector x[i] may be viewed as the "ith row" of the array. An associated location array loc(i), i = 1 to bNow, indicates the ranking of the solutions; that is,

[10] This section and the next are devoted explicitly to concerns of computer implementation. We include details that serve primarily for convenience as well as those that serve a more substantial function.

x[loc(1)] (the x vector stored in location "loc(1)") is the best solution, x[loc(2)] is the next best solution, and so forth.

A solution $x = x'$ is not permitted to be recorded if it duplicates another already in RefSet.[11] We speed the check to see if one solution duplicates another by keeping some simple auxiliary information about each solution. Specifically, in addition to keeping a value E(loc(i)) which identifies the evaluation (such as an objective function value) for x[loc(i)]), we may also keep a hash function value for this solution which we denote by Hash(loc(i)). Such a value can be determined by a single pass of the components of x[loc(i)].

To test for adding x' to RefSet, let E0 and Hash0 be defined for x' in the same way the corresponding quantities E(loc(i)) and Hash(loc(i)) are defined for x[loc(i)]. In addition to storing x' properly, if it qualifies to be added to RefSet, the RefSet Update Subroutine will keep track of a value NewRank, which will be the rank of x' (as 1st best, 2nd best, etc.) if it is added to RefSet, and otherwise receives the value 0. (Thus, checking whether NewRank is 0 or positive upon completion of the subroutine will automatically disclose whether x' succeeded or failed to qualify for inclusion.)

Finally, we let RefSetCall count the number of times the RefSet Update subroutine is called (which thus tells how many solutions are generated and examined as potential additions to RefSet), and let RefSetAdd count the number of times a solution is actually added. These values can be useful in the situation where it is desired to control the number of steps of executing the overall algorithm by limiting the maximum value of RefSetCall or RefSetAdd, for example. The values may appropriately be segregated and re-initialized to give separate statistics for the Initial Phase and for the Scatter Search/Path Relinking Phase of the SS/PR Template. Auxiliary information that may be useful is recorded by three counters, DupCheck, FullDupCheck and FullDupFound, which respectively count the number of partial duplication checks, full duplication checks, and the number of occurrences when duplications were found.

Pseudo-Code for the RefSet Update
Initialization Step

```
bNow = 0
RefSetCall = 0
RefSetAdd = 0
DupCheck = 0
FullDupCheck = 0
FullDupFound = 0
```

[11] Unlike the development of scatter search and path relinking, where duplications have no relevance, the notion of avoiding duplicate population members was not embraced in GA methods for many years, and some GA researchers continue instead to argue in favor of duplications (see, e.g., Goldberg, 1989). However, the work of Whitley and Kauth (1988) and Whitley (1989) introduces a "steady state" design that accords with the SS/PR perspective by seeking to avoid duplications, and the usefulness of this policy for such amended GAs has also been demonstrated by Davis (1991). There remain other types of duplications, however, that the GA orientation still does not contemplate, and which the SS/PR perspective identifies as important to eliminate – as discussed in Section 2.4 and in Appendix 2.

RefSet Update Subroutine (To Add x' to RefSet if it qualifies):
(This subroutine calls the Add Subroutine, specified below, which carries out the details of adding the solution x' to Refset.)
Begin Subroutine:

RefSetCall = RefSetCall + 1
NewRank = 0
If bNow = 0 then:
NewRank = 1
(this indicates that x' will be recorded as the solution with the first rank, i.e., the best, since no other solution is yet recorded.)
Compute Hash0
Call the Add Subroutine
End the RefSet Update Subroutine
Elseif bNow > 0 then:
(First check x' against the worst of the bNow best solutions, as follows:)
If E0 ≤ E(loc(bNow)) and if bNow = bMax then:
(x' is not better than the worst, and the list is full, so don't add x' to RefSet)
End the RefSet Update Subroutine.
Else
Compute Hash0
Endif
If E0 > E(loc(1)) then:
(x' is the new best solution of all, so record the rank of x' as NewRank)
NewRank = 1.
Else
(go through the solutions x[loc(1)] to x[loc(bNow)] in reverse order, to test if x' duplicates a previous solution or is new)
For i = bNow to 1 (decreasing index order)
If E(loc(i)) = E0 then:
DupCheck = DupCheck + 1
(this counts the duplication checks that are not trivially eliminated by the objective function value)
If Hash(loc(i)) = Hash0 then:
FullDupCheck = FullDupCheck + 1
(check x' against x[(loc(i)])
If x' = x[loc(i)] then:
FullDupFound = FullDupFound + 1
End the RefSet Update Subroutine
(x' is not added to RefSet)
Endif
Endif
Elseif E(loc(i)) > E0 then:
(by the sequence of the loop, the current i is the largest index i that satisfies E(loc(i)) > E0 – i.e., x[loc(i)] is the worst solution

that is still better than x', so x' will be ranked as the (i+1)-st best solution)
NewRank = i+1
Call the Add Subroutine
End the RefSet Update Subroutine
Endif
End i
(If the method reaches here, it has gone through the loop above without finding a solution better than x' and without finding a duplicate for x'. So x' qualifies as a new best solution, though its evaluation value must implicitly tie for best.)
NewRank = 1
Endif
Call the Add Subroutine
Endif
End RefSet Update Subroutine

Now we indicate the subroutine called by the RefSet Update Subroutine. In addition to the arrays already noted, we include a LastTime(loc0) array (updated at the end of the following Add Subroutine) where loc0 ranges over the locations loc(i), i = 1 to bMax. The array LastTime(loc0), which records the last ("most recent") time that the solution stored in location loc0 changes its identity, is important for linking with the routines of Section 4, which are designed to assure no duplicate subsets X of RefSet are ever generated).

Add Subroutine (to add x' to RefSet, given that it qualifies):
Begin Subroutine
RefSetAdd = RefSetAdd + 1
(x' will be recorded in the location occupied by the current worst solution. In case bNow < bMax, imagine the "worst" to be the empty solution in the location loc(bNow + 1), which will become loc(bNow) after incrementing bNow.)
If bNow < bMax then:
bNow = bNow + 1
loc(bNow) = bNow
Endif
(Next, the location pointers, loc(i), must be updated. First save the pointer to the solution that is currently worst, because this is the location where x' will be stored.)
loc0 = loc(bNow)
(Now update the location pointers that change, as a result of making x' the solution that acquires the rank of NewRank. We only need to change pointers from NewRank + 1 to bNow, because loc(NewRank) will be updated for x', below. To avoid destroying proper values, the change must be made in reverse order.)
If NewRank < bNow then:

```
        For i = bNow to NewRank + 1 (in decreasing index order)
            loc(i) = loc(i-1)
        End i
    Endif
    x[loc0] = x′
    (thus x′ is stored in the current location loc0)
    loc(NewRank) = loc0
    ( x′ will now be accessed as the solution whose rank is NewRank, via the loc
    array)
    Hash(loc0) = Hash0.
    (Finally, record the "time", given by NowTime, when the solution in location
    loc0 last changed its identity. NowTime is updated elsewhere, as shown later.)
    LastChange(loc0) = NowTime
End of Add Subroutine
```

A variation on ideas embodied in the preceding routines can be used to provide a method that checks for and eliminates duplications among solutions that are passed to the Improvement Method in Steps 2 and 7 of the SS/PR Template. By the philosophy of the approach developed here, such a weeding out of duplicates in the part of the overall approach can also be particularly useful, and a pseudo-code for the corresponding method is provided in Appendix 2.[12]

6 Choosing Subsets Of The Reference Solutions

We now introduce a special approach for creating different subsets X of RefSet, as a basis for implementing Step 5 of the SS/PR Template. It is important to note the SS/PR Template prescribes that the set C(X) of combined solutions (i.e., the set of all combined solutions that we intend to generate) is produced in its entirety at the point where X is created. Therefore, once a given subset X is created, there is no merit in creating it again. This creates a situation that differs noticeably from those considered in the context of genetic algorithms.

In some scatter search proposals, for example, the set C(X) associated with X consists of a single (weighted) center of gravity of the elements of X. Once the center of gravity is created from X, it is preferable to avoid recreating the same subset X in the future. Other proposals similarly specify a particular set of combined points to be created from a given subset X. These points may be variable in number, as from a deterministic algorithm applied to X that terminates when the quality of points generated falls below a threshold. However, the total number of such points that are

[12] A simple refinement can also be introduced in the current routine by identifying the first and last indexes of nonzero components of x' during the pass of the components of x' that identifies Hash0. In case the hash value check does not disclose that x' differs from x[loc(i)], the correspondence of these first and last indexes with those previously saved for x[loc(i)] can also be checked. This allows a full check of x' to x[loc(i)] to be restricted to components within the range of these indexes.

retained after an initial screening is usually small. Some path relinking proposals have a corresponding character (see, e.g., Glover, 1994b).

In such situations, we seek a procedure that generates subsets X of RefSet that have useful properties, while avoiding the duplication of subsets previously generated. Our approach for doing this is organized to generate four different collections of subsets of RefSet, which we refer to as SubSetType = 1, 2, 3 and 4. The principles we apply to generate these subsets can be applied to create additional of subsets of a related character. We can also adapt the basic ideas to handle a policy that, by contrast, allows selected subsets of solutions to be generated a second time — as where it may be desired to create more than one "brood" of offspring from a given collection of parents, under conditions where the history of the method suggests that such a collection should be singled out for generating additional offspring.

A central consideration is that RefSet itself will not be static, but will be changing as new solutions are added to replace old ones (when these new solutions qualify to be among the current b best solutions found). The dynamic nature of RefSet requires a method that is more subtle than one that itemizes various subsets of an unchanging RefSet. In addition, we wish to restrict attention to a relatively small number of subsets with useful features, since there are massive numbers of subsets that may be generated in general. The types of subsets we consider are as follows.

SubsetType = 1:	all 2-element subsets.
SubsetType = 2:	3-element subsets derived from the 2-element subsets by augmenting each 2-element subset to include the best solution not in this subset.
SubsetType = 3:	4-element subsets derived from the 3-element subsets by augmenting each 3-element subset to include the best solutions not in this subset.
SubsetType = 4:	the subsets consisting of the best i elements, for i = 5 to bNow.

The total number of subsets that satisfy the preceding stipulations is usually quite manageable. For example, if bMax = 10 there are 45 different 2-elements subsets for SubsetType = 1, and the collections for SubsetType = 2 and 3 each contain a bit less than 45 additional subsets. All together, SubsetType = 1 to 4 would generate approximately 130 distinct subsets. If bMax = 20, the total number of different subsets generated is a little less than 600. Depending on the number of solutions contained in C(X), and on the amount of time required to generate a given combined solution and to enhance it by the Improvement Method, the value of bMax can be increased or decreased, or the types of subsets produced can similarly be changed (to produce variants or subsets of the four types generated by the process subsequently indicated).

Since the method will continue to add new solutions to RefSet — until no new solutions can be found that are better than the bMax best — the number of subsets generated will typically be larger than the preceding figures. Consequently, a limit is placed on the number of solutions generated in total, in case the best solutions keep

changing. (A limit may also be placed on the number of iterations that elapse after one of the top 2 or 3 solutions has changed.) An appropriate cutoff can be selected by initial testing that gives the cutoff a large value, and by saving statistics to determine what smaller value is sufficient to generate good solutions.

Rationale

The reason for choosing the four indicated types of subsets of RefSet is as follows. First, 2-element subsets are the foundation of the first "provably optimal" procedures for generating constraint vector combinations in the surrogate constraint setting, whose ideas are the precursors of the ideas that became embodied in scatter search (see, e.g., Glover, 1965; Greenberg and Pierskalla, 1970). Also, conspicuously, 2-element combinations have for many years dominated the genetic algorithm literature (in "2-parent" combinations for crossover).

The generation of 2-element subsets also automatically generates (b-2)-element subsets, as complements of the 2-element subsets. We find the collection of (b-2)-element subsets less interesting, in the sense that the relative composition of such subsets tends to be much less varied than that of the 2-element subsets. (An exception occurs where the excluded elements may significantly affect the nature of the set C(X).)

Our extension of the 2-element subsets to 3-element subsets is motivated by the fact that the values 2 and 3 are (proportionally) somewhat different from each other, and hence we anticipate the 3-element subsets will have an influence that likewise is is somewhat different than that of the 2-element subsets. However, since the 3-element subsets are much more numerous than the 2-element subsets, we restrict consideration to those that always contains the best current solution in each such subset, which therefore creates approximately the same number of subsets as the 2-element subsets. A variation would be to replace this best solution with one of the top several best solutions, chosen pseudo randomly. Likewise, we extend the 3-element subsets to 4-element subsets for the same reason, and similarly restrict attention to a subcollection of these that always includes the two best solutions in each such subset. A simple alternative, which can be easily embedded in the framework we describe, is to generate all combinations of 3 solutions from the best 6 to be matched with each single remaining solution, or all combinations of 2 solutions from the best 4 or 5 to be matched with each pair of remaining solutions. Similar alternatives can be used where the subsets are restricted to 3 elements in total.

As the subsets become larger, the proportional difference in their successive sizes becomes smaller. The computational effort of handling such subsets also tends to grow. Hence we have chosen to limit the numbers of elements in these subsets (in this case to 4). Nevertheless, to obtain a limited sampling of subsets that contain larger numbers of solutions we include the special subsets designated as SubsetType = 4, which include the b best solutions as b ranges from 5 to bNow. (Recall that bNow increases as each new solution is added, until reaching bMax.) Since such subsets increasingly resemble each other for adjacent values of b as b grows, an option is to increment b by some fraction of bMax, e.g., bMax/5, instead of by 1.

6.1 Generating the Subsets of Reference Solutions

Each of the following algorithms embodies within it a step that consists of generating the set of combined solutions C(X) and executing the Improvement Method (e.g., as proposed in Section 4). Regardless of the form of the Improvement Method used, we will understand that the Reference Set Update method of the previous section is automatically applied with it.

To introduce the general approach to create the four types of subsets, we first briefly sketch a set of four corresponding simple algorithms that could be used in the situation where RefSet is entirely static (i.e., where the set of bMax best solutions never changes). These algorithms have the deficiency of potentially generating massive numbers of duplications if applied in the dynamic setting (where they must be re-initiated when RefSet becomes modified). However, their simple nature gives a basis for understanding the issues addressed by the more advanced approach.

In the following, when we specify that a particular solution is to become the second solution in X, we understand that the current first solution in X is unchanged, and similarly when we specify that a solution is to become the third (fourth) solution in X, we understand that the previous first and second (and third) solutions are unchanged.

Simple Algorithm for Subset Type 1

```
For i = 1 to bNow - 1
    Let x[loc(i)] be the first solution in X
    For j = i+1 to bNow
        Let x[loc(j)] be the second solution in X
        Create C(X) and execute the Improvement Method
    End j
End i
```

Simple Algorithm for Subset Type 2

```
Let x[loc(1)] be the first solution in X
For i = 2 to bNow - 1
    Let x[loc(i)] be the second solution in X
    For j = i+1 to bNow
        Let x[loc(j)] be the third solution in X
        Create C(X) and execute the Improvement Method
    End j
End i
```

Simple Algorithm For Subset Type 3
Let x[loc(1)] and x[loc(2)] be the first two solutions in X
For i = 3 to bNow - 1
    Let x[loc(i)] be the third solution in X
    For j = i+1 to bNow
        Let x[loc(j)] be the fourth solution in X
    Create C(X) and execute the Improvement Method
    End j
End i

Simple Algorithm for Subset Type 4
For i = 1 to bNow
    Let x[loc(i)] be the ith solution in X
    If i ≥ 5:
        Create C(X) and execute the Improvement Method
End i

6.2 Methods for a Dynamic RefSet

We must create somewhat more elaborate processes than the preceding to handle a dynamically changing reference set. We first indicate initializations that will be used to facilitate these processes.

Initialization Step:
For iLoc = 1 to bMax
    LastChange(iLoc) = 0
End iLoc
For SubsetType = 1 to 4
    LastRunTime(SubsetType) = 0
End SubsetType
NowTime = 0
StopCondition = 0
SubsetType = 0

In the iterative application of the steps of the SS/PR Template, we start from SubsetType = 0, as in the final statement of the Initialization Step, and repeatedly increase this index in a circular pattern that returns to SubsetType = 1 after reaching SubsetType = 4. However, the design that follows can be readily adapted to cycle through the subsets in any other order.

6.3 Arrays for the Subset Generation Method

There are two key arrays for the subset generation method, LastChange(loc(i)) and LastRunTime(SubsetType), that form the basis for avoiding duplications efficiently. We describe the function of these arrays, and the associated components that govern their updates, as follows.

(1) LastChange(loc(i)) – identifies the last (most recent) time that the solution stored in location loc(i) changed its identity (i.e., the last time a new solution was written over the old one in this location). More precisely, LastChange(loc(i)) is assigned the value NowTime when this change occurs. NowTime is increased by 1 each time one of the four algorithms prepares to generate the subsets of the type it deals with. As a result, NowTime is always 1 more than the value that could have been assigned to LastChange(loc(i)) on the previous execution of any (other) algorithm. Thus, the condition LastChange(loc(i)) = NowTime can only hold if a solution was changed in location loc(i) during the execution of the currrent algorithm for selecting elements to combine.

(2) LastRunTime(SubsetType) – identifies the last time (the value of NowTime on the last time) that the Algorithm SubsetType (= 1, 2, 3, or 4) was executed, prior to its current execution. Thus, for iLoc = loc(i), the condition LastChange(iLoc) < LastRunTime(SubsetType) means that the last time the solution x[iLoc] changed, occurred before the last time the Algorithm SubsetType was run. Hence x[iLoc] will now be the same as when the Algorithm looked at it previously. On the other hand, if LastChange(iLoc) ≥ LastRunTime(SubsetType), then the solution was changed either by Algorithm SubsetType itself, or by another algorithm executed more recently, and so x[iLoc] is not the same as when Algorithm SubsetType looked at it before.

6.4 Subset Generation Method

A basic part of the Subset Generation Method is the Subset Control Subroutine, which oversees the method and calls other subroutines to execute each Algorithm SubsetType (for SubsetType = 1 to 4). We indicate the form of this subroutine first. (The parameter StopCondition that governs the outer loop, which immediately follows, is initialized to 0 in the Initialization Step. When the cumulative number of executions of the Improvement Method, as it is applied within the various Algorithm Subroutines, exceeds a chosen limit, then StopCondition is set to 1 and the overall method thereby stops.)

Subset Control Subroutine

```
While StopCondition = 0 do
    SubsetType = SubsetType + 1
    If SubsetType > 4 then SubsetType = 1
    NowTime = NowTime + 1
    iNew = 0
    jOld = 0
```

(The next loop isolates all new (changed) solutions by storing their locations in LocNew(i), i = 1 to iNew, and all old (unchanged) solutions by storing their locations in LocOld(j), j = 1 to jOld. If iNew winds up 0, nothing has changed. When all algorithms are performed one after another, as here, regardless of sequence, the condition iNew = 0 means nothing has changed for any of them, and the method can stop.)

For i = 1 to bNow
iLoc = loc(i)
if LastChange(iLoc) ≥ LastRunTime(SubsetType) then
iNew = iNew + 1
LocNew(iNew) = iLoc
else
jOld = jOld + 1
LocOld(jOld) = iLoc
Endif
End i
If iNew = 0 then end the Subset Control Subroutine
(iNew = 0 here implies all combinations of the four types of subsets have been examined for their current composition without generating any new solutions, and so the SS/PR Template can terminate as a result of exhaustively considering all relevant subsets of RefSet in its final composition.)
If SubsetType = 1 Call Algorithm 1 Subroutine
If SubsetType = 2 Call Algorithm 2 Subroutine
If SubsetType = 3 Call Algorithm 3 Subroutine
If SubsetType = 4 Call Algorithm 4 Subroutine
(if StopCondition > 0 stop)
(Having identified the sets of old and new solutions and generated new combinations from them, update LastRunTime(SubsetType) to be the current NowTime value, so that the next time the algorithm is applied, LastRunTime(SubsetType) will be the last (most recent) time the algorithm was run.)
LastRunTime(SubsetType) = NowTime
End do
End Subset Control Subroutine

Next we identify the Algorithm Subroutines. Each Algorithm Subroutine works on the following principle. A subset X of RefSet can be new if and only if at least one element of X has not been contained in any previous X for the same SubsetType. We exploit this by sorting the solutions into old and new components, and executing a loop that first generates all combinations of new with new, and then a loop that generates all combinations of new with old. Meanwhile, any solution that is changed on the present application of the algorithm is excluded from being accessed once it has changed, because all subsets that include this solution will be generated on a later pass. To access a solution after it changes its rank, but before the loop is completed, would create duplications (unless the solution changes again), and in any case may

generate more solutions than necessary. The method generates the least number of solutions "currently known" to be necessary.

Algorithm 1 Subroutine
Begin Subroutine

```
(Currently iNew > 0. If iNew > 1, then look at all combinations of new with new)
If iNew > 1 then
    For i = 1 to  iNew - 1
        iLoc = LocNew(i)
        If LastChange(iLoc) < NowTime then
            (the solution in iLoc is still unchanged, so we can use it, otherwise
            the mehod ld skips it)
            Let x[iLoc] be the first element of X
            For j = i + 1 to iNew
                jLoc = LocNew(j)
                If LastChange(jLoc) < NowTime then
                    Let x[jLoc] be the second element of X
                    Create the set C(X) and execute the Improvement Method
                    (Optional check: if LastChange(iLoc) = NowTime, then
                    can jump to the end of the "I loop" to pick up the next I,
                    and generate fewer solutions.)
                Endif
            End j
        Endif
    End i
Endif
If jOld > 0 then
    For i = 1 to iNew
        iLoc =LocNew(i)
        If LastChange(iLoc) < NowTime then
            Let x[iLoc] be the first element of X
            For j = 1 to jOld
                jLoc = LocOld(j)
                If LastChange(jLoc) < NowTime then
                    Let x[jLoc] be the second element of X
                    Create the set C(X) and execute the Improvement Method
                    (Optional check: if LastChange(iLoc) = NowTime, then
                    can jump to the end of the "i loop" to pick up the next i,
                    and generate fewer solutions.)
                Endif
            End j
        Endif
    End i
Endif
```

End Subroutine

Algorithm 2 Subroutine
Begin Subroutine

```
loc1 = loc(1)
Let x[loc1] be the first element of X
If LastChange(loc1) ≥ LastRunTime(SubsetType) then
    (The solution in location loc1 is new, since last time Subroutine was run.)
    For i = 2 to  bNow - 1
        iLoc = loc(i)
        If LastChange(iLoc) < NowTime then
            (The solution in iLoc is still unchanged, so we can use it, otherwise
            the method skips it)
            x[iLoc] is the second element of X
            For j = i + 1 to bNow
                jLoc = loc(j)
                If LastChange(jLoc) < NowTime then
                    x[jLoc] is the third element of X
                    Create C(X) and execute the Improvement Method
                    (Optional check: if LastChange(iLoc) = NowTime, then
                    can jump to the end of the "i loop" to pick up the next i,
                    and generate fewer solutions.)
                Endif
            End j
        Endif
    End i
    End Algorithm 2 Subroutine (if reach here)
Else
    (The solution in location loc1 is not new, since last time.)
    (If iNew > 1, then look at all combinations of new with new.)
    If iNew > 1 then
        For i = 1 to  iNew - 1
            iLoc = LocNew(i)
            If LastChange(iLoc) < NowTime Then
                x[iLoc] is the second element of X
                For j = i + 1 to iNew
                    jLoc = LocNew(j)
                    If LastChange(jLoc) < NowTime then
                        x[jLoc] is the third element of X
                        Create C(X) and execute the Improvement Method
                        (Optional check: if LastChange(iLoc) = NowTime,
                        then can jump to the end of the "i loop" to pick up the
                        next i, and generate fewer solutions.)
                    Endif
                End j
            Endif
        End i
    Endif
```

```
        If jOld > 1 then
            For i = 1 to iNew
                iLoc =LocNew(i)
                If LastChange(iLoc) < NowTime then
                    Let x[iLoc] be the second element of X
                    For j = 2 to jOld
                        (loc1 is actually also LocOld(1))
                        jLoc = LocOld(j)
                        If LastChange(jLoc) < NowTime then
                            Let x[jLoc] be the third element of X
                            Create C(X) and execute the Improvement Method
                            (Optional check: if LastChange(iLoc) = NowTime,
                            then can jump to the end of the "i loop" to pick up the
                            next i, and generate fewer solutions.)
                        Endif
                    End j
                Endif
            End i
        Endif
    Endif
End Subroutine
```

The optional checks in Algorithms 1 and 2 are based on the fact that the condition LastChange(iLoc) = NowTime implies that x[iLoc] has been changed by finding a new solution with the Improvement Method. No duplications are created if the algorithm continues its course, using the old version of x[iLoc]. But it is also legitimate to jump to the end of the "i loop", as indicated, to generate fewer solutions. Algorithm 2 can also include a more influential check (in the same locations) which asks if LastChange(loc1) = NowTime, and terminates the current execution of the algorithm if so. In this case, a variation on the general organization could allow Algorithm 2 to be re-initiated immediately, since all it subsets will not incorporate a new "best overall" solution. Similar comments apply to introducing optional checks within the Algorithm 3 Subroutine, where LastChange(loc2) can also be checked. We do not bother to include further mention of such options.

```
Algorithm 3 Subroutine
Begin Subroutine
    loc1 = loc(1)
    loc2 = loc(2)
    Let x[loc1] and x[loc2] be the first two elements of X
    If LastChange(loc1) ≥ LastRunTime(SubsetType) or LastChange(loc2) ≥
    LastRunTime(SubsetType) then
        (The solution in location loc 1 or in loc2 is new, since last time.)
        For i = 3 to  bNow - 1
            iLoc = loc(i)
            If LastChange(iLoc) < NowTime Then
```

```
            (The solution in ILoc is still unchanged, so we can use it, otherwise
            the method skips it.)
            Let x[iLoc] be the third solution in X
            For j = i + 1 to bNow
                jLoc = loc(j)
                If LastTime(jLoc) < NowTime then
                    Let x[jLoc] be the fourth solution in X
                    Create C(X) and execute the Improvement Method
                Endif
            End j
        Endif
    End i
    End Algorithm 3 Subroutine (if reach here)
Else
    (Solutions in locations loc1 and loc2 are not new, since last time)
    (If iNew > 1, then we look at all combinations of new with new.)
    If iNew > 1 then
        For i = 1 to  iNew - 1
            iLoc = LocNew(i)
            If LastChange(iLoc) < NowTime Then
                Let x[iLoc] be the third solution in X
                For j = i + 1 to iNew
                    jLoc = LocNew(j)
                    If LastTime(jLoc) < NowTime then
                        Let x[jLoc] be the fourth solution in X
                        Create C(X) and execute the Improvement Method
                    Endif
                End j
            Endif
        End i
    Endif
    If jOld > 2 then
        For i = 1 to iNew
            iLoc =LocNew(i)
            If LastChange(iLoc) < NowTime then
                Let x[iLoc] be the third solution in X
                For j = 3 to jOld
                    jLoc = LocOld(j)
                    If LastChange(jLoc) < NowTime then
                        Let x[jLoc] be the fourth solution in X
                        Create C(X) and Execute Improvement Method
                    Endif
                End j
            Endif
        End i
    Endif
```

```
    Endif
End Subroutine

Algorithm 4 Subroutine
Begin Subroutine
    new = 0
    For i = 1 to 4
        iLoc = loc(i)
        Let x[iLoc] be the ith solution in X
        If LastChange(iLoc) ≥ LastRunTime(SubsetType) then new = 1
    End i
    For i = 5 to bNow
        iLoc = loc(i)
        If LastChange(iLoc) ≥ LastRunTime(SubsetType) then new = 1
        If LastChange(iLoc) < NowTime then
            Let x[iLoc] be the ith solution in X
            If new = 1 then
                Create C(X) and execute the Improvement Method
            Endif
        Endif
    End i
End Subroutine
```

The preceding subroutines complete the collection for generating subsets of the Reference Set, without duplication. The comments within the subroutines should be sufficient to make their rationale visible, and to provide a basis for variations of the forms previously discussed.

7 Improvement Method

We have already examined some of the issues relevant to designing an Improvement Method, particularly in reference to exploiting strongly determined and consistent variables. Such concerns underscore the importance of coordinating transition neighborhood methods with approaches that make use of constructive and destructive neighborhoods. Specifically, we re-emphasize that a process of exploiting strongly determined and consistent variables leads to generating and combining fragments of solutions, rather than complete solutions, and within such a setting the reliance on constructive and destructive processes is essential.

A first step toward considering an Improvement Method, therefore, entails an examination of elements appropriate to include in a constructive phase. (In the present discussion we bypass consideration of the simultaneous coordination of constructive and destructive processes, which is a theme of the tabu search component called strategic oscillation, and which is covered at length in the TS literature.)

A constructive phase that builds partial solutions into complete solutions is confronted by the need to decide among multiple alternative moves at each step. Often the relative attractiveness of such moves can be hard to differentiate except in a limited local sense. Under these circumstances it is appropriate either to scan portions of these alternatives in parallel, or else to iteratively generate more than one complete solution, as a basis for subsequent modification by a transition neighborhood approach. We first examine issues that are important for iterated constructive processes, and in later sections focus on parallel procedures, which are organized to be executed in a transition neighborhood setting as well as a constructive neighborhood setting.

In spite of their limitations, local evaluations generally reflect features that play a role in creating good solutions, and we are motivated to concentrate primarily on making moves with high evaluations. (In problem settings where the information content of such evaluations is low, and little meaning attaches to the difference between higher and lower evaluations, it is evidently appropriate to use revised evaluations that attenuate or otherwise amend original evaluations. In these cases, supplemental memory-based strategies for distinguishing among alternative choices become increasingly important.) A straightforward means for emphasizing high evaluations while simultaneously achieving variation in a constructive phase is either to adopt a probabilistic TS design, which isolates a subset of the top choices and weights them probabilistically as a monotone function of their evaluations, or to use an approach that alters the criteria for selecting moves on different passes in order to favor evaluations of varying ranks.[13]

An especially simple manifestation of the latter approach, which might be called a *variable-rank-choice* rule, selects only the best (highest evaluation) moves on one pass, then only the second best moves on another pass, and so on. This type of iterated approach can be carried beyond a constructive phase to launch a series of improving phases (where a kth best move, if not improving, is replaced by the current least improving move). The variable-rank-choice rule offers an experimental basis to uncover situations where the original evaluations may be misleading to various degrees, and to induce a useful variation or displacement of the choices under such circumstances. An apparent variant of this approach is to impose oscillation patterns on the choices, so that moves of various ranks are selected alternately, in specified proportions.

A variable-rank-choice is myopic, in the sense that it neglects several central considerations that can only be uncovered by a memory-based design. This shortcoming is illustrated by the situation in which choices available at one stage of a constructive (or transition) process persist for a number of moves, and remain among the attractive choices. When this occurs, a policy which periodically (or uniformly)

---

[13] The GRASP procedure, which has gained some popularity in recent years, uses the special variant of a probabilistic TS design where the probability assigned to every member of the chosen subset is the same; i.e., the choice among these members is random. This strategy is applied in GRASP without reference to memory or search history, and strictly within the confines of a constructive phase. See, e.g., Feo and Resende (1995).

chooses second or third best moves may not appreciably change the solution that results by choosing only the "first best" moves, because the second and third best moves may also correspond to choices destined to become first best choices on later iterations (or that were already first best choices on previous iterations). An appropriate reliance on memory can identify such phenomena, and provide a foundation for making compensating choices that will assure suitable variability.

Likewise, variable-rank-choice is blind to conditional effects that can be advantageously exploited by the form of adaptive memory incorporated in TS, as where a particular choice remains attractive for several steps before it finally becomes ranked highly enough to be chosen. Given the eventual choice of this move, a revised sequence of choices my prove advisable, which selects the move at an earlier stage, and therefore gains the benefit of disclosing modified evaluations of other moves that may uncover a different set of preferred choices.

In reverse, attractive moves that are not executed, and which become inaccessible or unattractive later, give evidence of alternatives that provide access to somewhat different regions of the search space. Tracking the occurrence of unselected moves of this type gives a means for identifying alternatives that are useful for achieving diversification. In a related manner, a record of unselected moves that persistently or recurrently become attractive, over a limited but non-negligible interval, provides a type of strategy whose relevance is stressed in the TS literature, but which also is often neglected. Such considerations can readily be integrated with the critical event memory approach described in section 2.3, which takes account of solutions previously generated at critical events. The indicated relationships between evaluations and choices that can be exploited by historical monitoring are the type that are particularly susceptible to being analyzed by the tool of target analysis (Glover and Laguna, 1997).

While we have touched only cursorily on such alternatives for creating improved constructive procedures, we emphasize their relevance for processes that build fragments of solutions into complete solutions during the application of an SS/PR approach. In the sections that follow, we turn to a consideration of elements that are important to the creation of an Improvement Method at stages beyond those primarily devoted to construction.

7.1 A Filter and Fan Method

There often exist alternative neighborhoods of moves available to compose improvement methods for various kinds of optimization problems. Experience from numerous applications suggests that there is merit in using more than one such neighborhood. For example, a common theme of strategic oscillation is to cycle among alternative neighborhoods according to various patterns. Strategic oscillation also commonly operates by cycling through various regions, or “levels” of a given neighborhood.

The approach of cycling through different levels of a neighborhood is manifest in two types of candidate list strategies, the Filtration Strategy and the Sequential Fan

Strategy, proposed with tabu search (see, e.g., Glover and Laguna, 1997). The goal of these strategies is to identify attractive moves with an economical degree of effort. In addition, however, the Filtration and Sequential Fan strategies offer a useful basis for converting a simple Improvement Method into a more advanced one. We propose a way to marry these two candidate list strategies to create a *Filter and Fan Method* which provides a convenient form of an Improvement Method for the SS/PR Template.

We have selected a type of improvement approach that has the convenient feature of being able to extend or enhance other improvement procedures that may have independently demonstrated their utility. For example, components of other procedures – such as the moves they rely on and the evaluations they use to choose among these moves – can be embedded within the following general approach in an entirely straightforward manner.

Component Moves

The moves to serve as building blocks for the proposed method will characteristically be simple types of moves as illustrated by adjacent integer changes for integer variables (e.g., "flip" moves for 0-1 variables) and by elementary insert or swap moves for permutation problems. We call the chosen component moves level 1 moves, or *1-moves*.

An associated Level 1 Improvement Method can be defined relative to the 1-moves, which operates by segregating a collection of 1-moves by a preliminary candidate list approach, such as an Aspiration Plus strategy (Glover and Laguna, 1997). A random selection of such a collection is possible, in the interest of simplification, but at an appreciable risk of reducing overall effectiveness. (When randomization is used, the initial list should typically be larger than otherwise required.)

A useful goal for the initial candidate list strategy is to assure that a number of these 1-moves are among the highest evaluation moves currently available, so that if none of them is improving, the method is likely to be at a local optimum relative to these moves. The Level 1 Method then terminates when no moves from its candidate list are improving moves, thus presumably stopping at a local optimum or a "near" local optimum (relative to a larger collection of moves that encompasses those of the candidate list). The candidate list construction for Level 1 can be dynamic to allow the size of the list to grow when no improving move is found. (The Aspiration Plus strategy has this character, for example.) Such an optimum makes it possible to assure that termination will occur at a local optimum, if desired. The Filter and Fan Method then goes beyond this stopping point to create higher level moves. For this purpose, we extract a subset M of some number of best moves from those examined by the Level 1 method when it terminates, where for example $|M| = 20$ or 40.

General Design.

The general design of the Filter and Fan Method is to isolate a subset M(L) of the best moves at a given level L, to be used as a basis for generating more advanced

moves at level L+1 when level L fails to yield an improving move. In case L=1, we choose M(1) to be a subset of M.

Suppose that m is a given L-move from M(L), and let A(m), be a related set of 1-moves (derived from M) so that the result of applying any 1-move m' in A(m), after applying m will create an (L+1)-move which we denote by m@m'. By restricting M(L) to consist of a relatively small number of the moves examined at Level L (e.g., choosing |M(L)| = 10 or 20), and likewise restricting A(m) to consist of a relatively small number of 1-moves, the total number of L+1 moves m@m' can be maintained at a modest size. For example, a steady state choice that always picks |M(L)| = 16 and |A(m)| = 8 (for each m in M(L)) will generate only 128 (L+1)-moves to be examined at each level L+1. If none are improving, in this example the 16 best are selected to compose M(L+1), and the process repeats.

The utility of this design is to avoid the combinatorial explosion of possibilities that results by generating the set of all possible (L+1)-moves at each step. Instead the approach filters a subset M(L) of best moves at level L, and for each of these moves likewise filters a set A(m) of best 1-moves from M. The "fan" that generates |M(L)||A(m)| potential (L+1)-moves as candidates to examine at Level L+1 is therefore maintained to be of reasonable size.[14]

We say that A(m) is *derived from* M, not only because A(m) may be smaller than M, but also because some of the moves of M may not be legitimate once the L-move m in M(L) is executed. The 1-moves available after applying move m may not precisely correspond to moves of the original set M. For example, if the 1-moves correspond to flipping the values of 0-1 variables, then a move m may have flipped values for several variables in M, and the corresponding 1-moves in M will no longer be accessible. However, it is generally easy to keep a record for each move m in M(L) that identifies the moves of M that should be excluded from A(m), allowing A(m) to be composed of the best |A(m)| remaining members of M. Similar comments apply to moves such as swap moves and insert moves.

A simple steady state version of the Filter and Fan method can be summarized as follows. Let n0 be the chosen size of the initial M, n1 be the size of each set M(L) and n2 be the size of each set A(m) (where n1 and n2 do not exceed n0). In many applications, n0 will be at most 40 and n1 and n2 will be at most 20 (and smaller values may be preferable). We call this version a *strict* improvement method because it does not allow steps that are nonimproving, and therefore terminates at a local optimum relative to the multilevel moves it employs.

---

[14] The emphasis on controlling computational effort while producing good candidate moves can be facilitated in some settings by an accelerated (shortcut) evaluation process. This can occur by selecting members of an initial candidate list that provides the source of M, as illustrated by the use of surrogate constraint evaluations in place of a lengthier evaluation that identifies the full consequences of a move relative to all constraints of a problem. Accelerated evaluations can also be applied to isolating M from the initial candidate list, while reserving more extensive types of evaluations to isolating M(1) from M, and to deriving A(m) from M for the moves m generated at various levels.

Filter and Fan Strict Improvement Method

1. *Generate a candidate list of 1-moves for the current solution x.*
   (a) If any of the 1-moves are improving: Choose the best member from the list and execute it to create a new current solution x. (The "best member" may be the only member if the list terminates with the first improving move encountered.) Then return to the start of step 1.
   (b) If none of the 1-moves are improving: Identify the set M of the n0 best 1-moves examined. Let M(1) be a subset of the n1 best moves from M, and let X(1) be the set of solutions produced by these moves. Set L = 1 and proceed to step 2.
2. *For each L-move m in M(L)*: Identify the associated set A(m) of the n2 best compatible moves m' derived from M, and evaluate each resulting (L+1)-move m@m'. (Equivalently, evaluate each solution that results by applying move m' to the corresponding member of X(L).) When fewer than n2 moves of M are compatible with move m, restrict consideration to this smaller set of moves in composing A(m).
   (a) If an improving move is found during the foregoing process: Select the best such move generated (by the point where the process is elected to terminate), and execute the move to create a new current solution x. Then return to step 1.
   (b) If no improving move is found by the time all moves in M(L) are examined: Stop if L has reached a chosen upper limit MaxL. Otherwise, identify the set M(L+1) of the n1 best (L+1)-moves evaluated (and/or identify the associated set X(L+1)). (If fewer than n1 distinct (L+1)-moves are available to be evaluated, then include all distinct (L+1)-moves in M(L+1).) Then set L = L + 1 and return to the start of Step 2.

The identification of M(L+1) (and/or X(L+1)) in Step 2(b) can of course be undertaken as part of the process of looking for an improving move, rather than waiting until no improving move is found. The appropriate organization depends on the setting. Also, an evident option for executing the preceding method is to allow a variable-state version where n1 and/or n2 decreases as L increases, thereby reducing the number of candidates for successively higher levels of moves. Another option is to allow L to change its value by a different pattern. We now examine relevant considerations for implementing this method.

7.2 Avoiding Duplications

A slight change in the preceding description can improve the approach by avoiding the generation of a number of duplicate outcomes. Such duplications are especially likely to arise when generating 2-moves, in the setting where solutions are generated by flipping 0-1 variables. To illustrate, suppose n0 = n1 = n2 = 20. Thus, initially M consists of the 20 best 0-1 flips, and we consider the case where none are improving. Then at Step 2, having chosen M(1) = M (since n1 = n0), the method as described

would select each move m from M(1) and extend it by applying a compatible 1-move m' taken from A(m), where in this case A(m) consists of all of M excluding the single flip that produced m. Thus, each move m in M(1) would be matched with the 19 other 1-moves that constitute flips other than the one embodied in m. Over the 20 elements of M(1) this yields 20X19 = 380 possibilities to evaluate. But this is double the number that is relevant, since each flip of two variables x_i and x_j will be generated twice — once when the x_i flip is in M(1) and the x_j flip is in A(m), and once when the x_j flip is in M(1) and the x_i flip is in A(m). Such duplication can easily be removed by restricting a flip of two variables x_i and x_j so that j > i, where x_i belongs to M(1) and x_j belongs to A(m). This indexing restriction may alternately be applied after the flips are sorted in order of their attractiveness. In either case, the result may be viewed as restricting the definition of A(m).

Potential duplications for other types of moves can similarly be easily avoided at the level L = 1, where 2-moves are being generated in Step 2. In the case of swap and insert moves, a more balanced set of options can be created by restricting the number of moves recorded in M that involve any given element (or position). For example, if 5 of the 20 best swap moves involve swapping a given element i, then it may be preferable to record only the 2 or 3 best of these in M, and therefore complete the remainder of M with moves that may not strictly be among the 20 best.

As L grows larger, the chance for duplications drops significantly, provided they have been eliminated at the first execution of Step 2. Consequently, special restrictions for larger values of L can be disregarded. Instead, it is easier to screen for duplications at the point where a move becomes a candidate to include in M(L+1) (or equivalently, the associated solution becomes a candidate to include in X(L+1)). The method for updating the Reference Set, given in the Section 3, can be used as a design to conveniently identify and eliminate such duplications in the present context as well.[15]

7.3 Move Descriptions

The influence of move descriptions, where the same move can be characterized in alternative ways, can affect the nature of moves available independently of the neighborhood used. This phenomenon is worth noting, because standard analyses tend to conceive the neighborhood structure as the sole determinant of relevant outcomes. The phenomenon arises in the Filter and Fan method because the move description implicitly transmits restrictions on deeper level moves in a manner similar to the imposition of *tabu* restrictions in tabu search. Thus, an implicit memory

[15] This can be made simpler by recording "incremental solutions" – or more precisely the representations of solutions in X(L) that result when the current solution x is represented as the zero vector – since these will generally have few nonzero components, and may be stored and accessed more quickly than complete solution vectors.

operates by means of the attributes of the moves considered, and these attributes depend on the move description.

To illustrate, a description that characterizes an insert move to consist of inserting element i in position p (and shifting other elements appropriately) can yield a different outcome, once intervening moves are made, than a description that characterizes the same move as inserting element i immediately before an element v. (Note that a more restrictive description, such as specifying that the move consists of inserting element i between elements u and v, may render the move impossible once either u or v changes its position.)

The phenomenon can be usefully demonstrated for swap moves by the situation where two moves in M are respectively characterized as swapping elements i and j and swapping elements i and k. After performing the first swap, the second will receive a changed evaluation, since i is no longer in the same position. If instead the same moves are characterized as swapping the elements in positions p and q, and in positions p and r (where elements i, j and k are currently in these positions), then the result of the first swap gives a different outcome for the second swap than the one illustrated previously; that is, the second swap now corresponds to swapping elements j and k rather than i and k. (Still another outcome results if a swap is characterized as a double insert move, e.g., as inserting an element i immediately after the current predecessor of j and inserting j immediately after the current predecessor of i.)

A preferable description of course depends in part on the nature of the problem. An interesting possibility is to allow two (or more) move characterizations, and then to choose the one in a given situation that yields the best result. This is analogous to allowing different solution attributes to define potential restrictions by the attribute-based memory of tabu search.

By the same token, it may be seen that greater flexibility can be obtained simply by relaxing the definition of a move. For example, in the setting of 0-1 problems, instead of characterizing M as a set of value-specific flips (as illustrated by stipulating that x_j should change from 0 to 1), we can allow M to be value-independent (as illustrated by stipulating that x_j should change to 1 - x_j). The value -independent characterization allows greater latitude for generating moves from M, and is relevant as L grows beyond the value of 1. Such a characterization should be accompanied by a more explicit (and structured) use of tabu search memory, however, to control the possibility of cycling. The value-specific characterization is sufficiently limiting to avoid this need in the present illustration.

Another degree of latitude exists in deriving A(m) from M. Suppose the moves of M are denoted m(1), m(2), ..., m(u), where the moves with smaller indexes have higher evaluations. If we stipulate that A(m) should consist of the r best of these moves, restricted to those that are compatible with m, we may choose to avoid some computation by ordering M in advance, as indicated, and then simply selecting the first r compatible members to compose A(m). However, since the relative attractiveness of the moves in M may change once the move m is made, an alternative strategy is instead to examine some larger number of compatible members m' of M to

improve the likelihood of including the “true r best” options for the compound moves m@m'. This of course does not change the form of the method, since it amounts to another possibility for choosing the size of |A(m)|. However, this observation discloses that it may be preferable to choose the size of |A(m)| to be larger relative to the size of |M(L)| than intuition may at first suggest.

7.4 Definitions of 1-moves and the Composition of M

It is entirely possible in the application of the Filter and Fan Method that a 1-move may be defined in such a way to produce an infeasible solution, but a coordinated succession of 1-moves will restore feasibility. A simple example is provided by the graph partitioning problem, where feasible solutions consist of partitioning a set of 2n nodes into two sets that contain n nodes each. A 1-move that consists of swapping two nodes that lie in different sets will maintain feasibility, but a 1-move that consists of moving a node from one set to the other will not. Nevertheless, since the second (simpler) 1-moves are many fewer in number, a candidate list strategy that identifies attractive moves of this type, differentiated according to the set in which they originate, also provides a useful basis for composing compound moves by the Filter and Fan Method. In this situation, the successive 1-moves must alternately be chosen from different sets, and feasible solutions only occur for even values of L. Such implementations are easily accommodated within the general framework, and we do not bother to introduce more complicated notation to represent them.

Similarly, a method that applies 0-1 flips to an integer programming problem may allow M to include flips that create infeasibility (as in a muldimensional knapsack problem where any flip from 0 to 1 will drive a “boundary solution” infeasible). Rather than avoiding moves that produce infeasibility, the process may make provision for an infeasibility generated at a level L to be countered by focusing on moves to recover feasibility at level L + 1. (The focus may be extended to additional levels as necessary.)

A related but more advanced situation arises in ejection chain strategies where 1-moves may be defined so that no sequence of them produces a feasible solution. In this type of construction a reference structure guides the moves selected to assure that a feasible solution can always be generated by one or more associated trial solutions (see, e.g., Glover, 1992; Rego 1996; Rego and Roucairol, 1996). In these instances the Filter and Fan Method can accordingly be modified to rely on the trial solutions as a basis for evaluating the L-moves generated.

The composition of M can be affected by an interdependency among subcollections of moves. In certain pivot strategies for network flow optimization, for example, once the best pivot move associated with a given node of the network is executed, then the quality of all other moves associated with the node deteriorates. Thus, instead of choosing M to consist of the n0 best moves overall, which could possibly include several moves associated with the same node, it can be preferable to allow M to include only one move for any given node. More generally, M may be restricted by limiting the numbers of moves of different categories that compose it.

The potential to modify M, M(L) and A(m) as the method progresses can be expanded by allowing these sets to be "refreshed" by a more extensive examination of alternatives than those earmarked for consideration when L = 1. For example, in some settings the execution of a move m may lead to identifying a somewhat restricted subset of further moves that comprise the only alternatives from which a compound improving move can be generated at the next level. In such a case, M and A(m) should draw from such alternatives even though they may not be encompassed by the original M. Such an expansion of M and its derivative sets must be carefully controlled to avoid losing the benefit of reducing overall computation that is provided by a more restricted determination of these sets.

Allowing for such an expansion can increase the appropriate value of MaxL. Or inversely, the reliance on smaller sizes of M, M(L) and A(m) can decrease the appropriate value of MaxL. A reasonable alternative is to determine MaxL indirectly by a decision to terminate when the quality of the best current L-moves (or of the 1-moves that are used to generate these L-moves) drops below a chosen threshold. These considerations are relevant to the "partial refreshing" process of the Multi-Stream variant, described in the next section.

7.5 Advanced Improvement Alternatives

The preceding Strict Improvement version of the Filter and Fan method has a conspicuous limitation that can impair the quality of the solutions it produces. In the absence of applying a refreshing option, which can entail a considerable increase in computation or complexity, the merit of the set M as the source of the sets M(L) and A(m) diminishes as L grows. This is due to the fact that M is composed of moves that received a high evaluation relative to the current solution before the compounding effects of successive 1-moves is considered, and as these moves are drawn from M for progressively larger values of L, the relevance of M as the source of such moves deteriorates. Under such circumstances the value of MaxL will normally not be usefully chosen to be very large (and a value as small as 4 or 5 may not be atypical).

This limitation of the strict improvement method can be countered by one of three relatively simple schemes:

(1) A Shift-and-Update variant.
(2) A Diversification variant.
(3) A Multi-Stream variant.

We examine these three variants as follows.

Shift-and-Update Variant

The Shift-and-Update method operates by choosing a nonimproving move when the method would otherwise terminate, but shifting the outcome away from the lowest levels to insure that the choice will yield a new solution that lies some minimum number of 1-moves "away from" the current solution x. The new solution is then

updated to become the current solution, and the method repeats. The method can be described by the following rule.

Shift-and-Update Rule

> Identify a value MinL such that $1 < \text{MinL} \leq \text{MaxL}$. When no improvement is found upon reaching L = MaxL, select the best solution found over the levels $L \geq \text{MinL}$. Then specify this solution to be the current solution x and return to the beginning of Step 1.

Simple recency-based tabu search memory can be used with this rule to keep from returning to a preceding solution and generally to induce successively generated solutions to continue to move away from regions previously visited. This variant creates a degree of diversification which may be increased by restricting attention to value-specific move descriptions, so that each set M(L) will be progressively farther removed from M(1).

Diversification Variant

A more ambitious form of diversification, which can be applied independently or introduced as a periodic alternative to the Shift-and-Update approach, operates by temporarily replacing the steps of the Filter and Fan Method with a series of steps specifically designed to move away from regions previously visited. We express this approach by the following rule.

Diversification Rule

> Introduce an explicit Diversification Stage when the approach fails to find an improvement (and terminates with L = MaxL). In this stage change the evaluator to favor new solutions whose attributes differ from those of solutions previously encountered, and execute a chosen number MinL of successive 1-moves guided by the changed evaluator. Immediately following this stage, the next pass of the Filter and Fan Method (starting again from Step 1 with a new x) uses the normal evaluator, without a diversification influence.

This diversification approach can use tabu search frequency memory to encourage moves that introduce attributes that were rarely or never found in solutions previously generated, and similarly to encourage moves that eliminate attributes from the current solution that were often found in previous solutions. An alternative is to apply the Diversification Stage directly within the Filter and Fan structure, where M and its derivative sets are generated by keeping the value of n0, and especially the values n1 and n2, small. The evaluator in this case may alternatively be a composite of the normal evaluator and a diversification evaluator, with the goal of producing solutions that are reasonably good as well as somewhat different from previous solutions. Then the values of n0, n1 and n2 may be closer to those during a non-diversification stage. For a simpler implementation, instead of using frequency memory in the

Diversification Stage, the approach can be applied as noted in Section 1 by introducing a Post-Improvement step that uses the Diversification Generator.

Multi-Stream Variant

The Sequential Fan candidate list strategy embodies an additional element, not yet considered, that consists of generating several solution streams simultaneously. The inclusion of this element in the Filter and Fan Method produces a method that increases the likelihood of finding improved solutions. The ideas underlying this variant may be sketched in overview as follows.

Starting from the current solution x in Step 1 of the Filter and Fan Method, the approach selects a small number s of the best moves from the set M. Each selected move is allowed to initiate a different stream by creating an associated set of additional moves to launch the rest of the Filter and Fan Method. The outcome effectively shifts the Filter and Fan Method by one step, so that the solutions produced by these s moves take the place of solution x to produce additional levels of moves.

Specifically, let x[i], $i \in S = \{1,...,s\}$ denote the solutions produced by the s best moves derived from x. The evaluations for each x[i] are updated to generate a refreshed candidate list for each. Let M(0:i) denote the set of n0 best 1-moves from the candidate list created for x[i] (in the same way that M represents the set of n0 best 1-moves from the candidate list created for x). Similarly, let M(L:i) and X(L:i), $i \in S$, denote the sets of moves and corresponding sets of solutions for an arbitrary level $L \geq 1$.

The Multi-Stream variant then operates exactly as the single stream variant, by allowing each x[i] for $i \in S$ to take the role of x, except that the computation is restricted in a special way. The purpose of this restriction is to allow the Multi-Stream variant to require only slightly more effort than the single stream variant, except for the initial step that identifies a fresh candidate list for each x[i]. (To achieve greater speed, an option is to forego this identification and instead rely on choosing each M(0:i) as a subset of an enlarged set M generated for x. The relevance of this alternative is determined by considerations discussed in Section 3.)

The restricted treatment of the sets M(L:i) at each stage of the Multi-Stream variant produces the following modifications of Steps 1 and 2 of the Filter and Fan Method.

Multi-Stream Implementation

1A. (*Modification of step 1.*)

(a) As in the original step 1(a), return to the start of step 1 with a new solution x if an improving move is identified while examining the candidate list for x. Similarly, if no such improving move is found, but an improving move is identified in the process of examining the candidate list for one of the solutions x[i], $i \in S$, then choose the best such move (up to the point where the process is elected to discontinue), and return to the start of step 1.

(b) If no improving moves are found in step 1A (a), then amend the approach of the original step 1 (b) as follows. Instead of creating each set M(1:i) as the n1 best moves from M(0:i), coordinate the streams so that the entire collection M(1:i), $i \in S$, retains only the n1 best moves from the collection M(0:i), $i \in S$. (Thus, on average, each M(1:i) contains only n1/s moves. Some of these sets may be empty.) Then set L = 1 and proceed to step 2A.

2A. (*Modification of step 2*). Consider each $i \in S$ such that M(L:i) is not empty. For each L-move in M(L:i) identify the associated set A(m) of the n2 best compatible 1-moves taken from M(0:i), to produce candidate (L+1)-moves of the form m@m' for $m' \in A(m)$.

(a) If an improving (L+1)-move is found, select such a move as in the original step 2(a) and return to the start of step 1 with a new x.

(b) If no improving move is found while examining the set of moves in the collection M(L:i), $i \in S$: Stop if L has reached MaxL. Otherwise, identify the collection M(L+1:i), $i \in S$, to consist of the best n1 moves generated from the entire collection M(L:i), $i \in S$. Then set L = L + 1 and return to the start of step 2A.

The Multi-Stream variant requires some monitoring to assure that the sets M(L:i) do not contain duplicate L-moves (i.e., the sets X(L:i) do not contain duplicate solutions), or to assure that such duplications are removed when they occur. Simple forms of TS memory can be used for this purpose, or again the type of screening used to update the set of Reference Solutions for the SS/PR Template (as described in Section 3) can be employed with the method. Special restrictions for level L = 1, as previously noted for the single-stream case discussed in the first part of Section 5, can be readily extended to the Multi-Stream case.

The Multi-Stream approach can be joined with the Shift-and-Update variant or the Diversification variant, and is particularly susceptible to being exploited by parallel processing.

8 Conclusions

This paper is intended to support the development of scatter search and path relinking methods, by offering highly specific procedures for executing component routines. Included are proposals for new diversification generators and for special processes to avoid generating or incorporating duplicate solutions at various stages. These developments point to the relevance of including adaptive memory designs of the form proposed in connection with tabu search. We have also undertaken to indicate a type of Improvement Method that exploits the marriage of two TS candidate list strategies – the Filtration and Sequential Fan strategies – that invite further examination in their own right. While there are additional ways to implement scatter search and path relinking, the SS/PR Template and its subroutines offer a potential to

facilitate the creation of initial methods and to reduce the effort involved in creating additional refinements.

REFERENCES

1. Consiglio, A. and S.A. Zenios (1996). "Designing Portfolios of Financial Products via Integrated Simulation and Optimization Models," Report 96-05, Department of Public and Business Administration, University of Cyprus, Nicosia, CYPRUS, to appear in *Operations Research.* [http://zeus.cc.ucy.ac.cy/ucy/pba/zenios/public.html]
2. Consiglio, A. and S.A. Zenios (1997). "a Model for Designing Callable Bonds and its Solution Using Tabu Search." *Journal of Economic Dynamics and Control* 21, 1445-1470. [http://zeus.cc.ucy.ac.cy/ucy/pba/zenios/public.html]
3. Crowston, W.B., F. Glover, G.L.Thompson and J.D. Trawick (1963). "Probabilistic and Parametric Learning Combinations of Local Job Shop Scheduling Rules," ONR Research Memorandum No. 117, GSIA, Carnegie Mellon University, Pittsburgh, PA
4. Cung, V-D., T. Mautor, P. Michelon, A. Tavares (1996). "Scatter Search for the Quadratic Assignment Problem", Laboratoire PRiSM-CNRS URA 1525. [http://www.prism.uvsq.fr/public/vdc/CONFS/ieee_icec97.ps.Z]
5. Davis, L., ed. (1991). *Handbook of Genetic Algorithms*, Van Nostrand Reinhold.
6. Feo, T.A. and M.G.C. Resende, (1995). "Greedy Randomized Adoptive Search Procedures," *Journal of Global Optimization*, 6, 109-133.
7. Fleurent, C., F. Glover, P. Michelon and Z. Valli (1996). "A Scatter Search Approach for Unconstrained Continuous Optimization," *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, 643-648.
8. Freville, A. and G. Plateau (1986). "Heuristics and Reduction Methods for Multiple Constraint 0-1 Linear Programming Problems," *European Journal of Operational Research*, 24, 206-215.
9. Freville, A. and G. Plateau (1993). "An Exact Search for the Solution of the Surrogate Dual of the 0-1 Bidimensional Knapsack Problem," *European Journal of Operational Research*, 68, 413-421.
10. Glover, F. (1963). "Parametric Combinations of Local Job Shop Rules," Chapter IV, ONR Research Memorandum no. 117, GSIA, Carnegie Mellon University, Pittsburgh, PA.
11. Glover, F. (1965). "A Multiphase Dual Algorithm for the Zero-One Integer Programming Problem," *Operations Research*, Vol 13, No 6, 879.
12. Glover, F. (1975). "Surrogate Constraint Duality in Mathematical Programming," *Operations Research*, 23, 434-451.
13. Glover, F. (1977). "Heuristics for Integer Programming Using Surrogate Constraints," *Decision Sciences*, Vol 8, No 1, 156-166.
14. Glover, F. (1992). "Ejection Chains, Reference Structures and Alternating Path Methods for Traveling Salesman Problems," University of Colorado. Shortened version published in *Discrete Applied Mathematics*, 1996, 65, 223-253. [http://spot.colorado.edu/~glover (under Publications)]
15. Glover, F. (1994a). "Genetic Algorithms and Scatter Search: Unsuspected Potentials," S*tatistics and Computing*, 4, 131-140. [http://spot.colorado.edu/~glover (under Publications)]
16. Glover, F. (1994b). "Tabu Search for Nonlinear and Parametric Optimization (with Links to Genetic Algorithms)," *Discrete Applied Mathematics*, 49, 231-255. [http://spot.colorado.edu/~glover (under Publications)]
17. Glover, F. (1995). "Scatter Search and Star-Paths: Beyond the Genetic Metaphor," *OR Spectrum*, 17, 125-137. [http://spot.colorado.edu/~glover (under Publications)]
18. Glover, F., J. P. Kelly and M. Laguna (1996). "New Advances and Applications of Combining Simulation and Optimization," *Proceedings of the 1996 Winter Simulation*

Conference, J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain (Eds.), 144-152. [http://spot.colorado.edu/~glover (under OptQuest heading)]

19. Glover, F. and M. Laguna (1997). *Tabu Search*, Kluwer Academic Publishers. [http://spot.colorado.edu/~glover (under Tabu Search heading)]
20. Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*, Reading, Massachusetts: Addison-Wesley.
21. Greenberg, H. J. and Pierskalla, W.P. (1970). "Surrogate Mathematical Programs," *Operations Research*, 18, 924-939.
22. Holland, J.H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI.
23. Karwan, M.H. and R.L. Rardin (1976). "Surrogate Dual Multiplier Search Procedures in Integer Programming," School of Industrial Systems Engineering, Report Series No. J-77-13, Georgia Institute of Technology.
24. Karwan, M.H. and R.L. Rardin (1979). "Some Relationships Between Lagrangean and Surrogate Duality in Integer Programming," *Mathematical Programming*, 17, 230-334.
25. Kelly, J., B. Rangaswamy and J. Xu (1996). "A Scatter Search-Based Learning Algorithm for Neural Network Training," *Journal of Heuristics*, Vol. 2, pp. 129-146.
26. Laguna, M. and R. Marti (1997). "GRASP and Path Relinking for 2-Layer Straight Line Crossing Minimization," Research Report, University of Colorado [http://www.bus.colorado.edu/Faculty/Laguna/Papers/crossmin.html]
27. Laguna, M. (1997). "Optimizing Complex Systems with OptQuest," Research Report, University of Colorado, [http://www.bus.colorado.edu/Faculty/Laguna/Papers]
28. Laguna, M., R. Martí and V. Campos (1997). "Tabu Search with Path Relinking for the Linear Ordering Problem," Research Report, University of Colorado. [http://www.bus.colorado.edu/Faculty/Laguna/Papers/lop.html]
29. Muhlenbein, H. (1997). "The Equation for the Response to Selection and its Use for Prediction," to appear in *Evolutionary Computation*. [http://set.gmd.de/AS/ga/ga.html]
30. Rana, S. and D. Whitley (1997). "Bit Representations with a Twist," Proc. 7th International Conference on Genetic Algorithms, T. Baeck ed. pp: 188-196, Morgan Kaufman. [http://www.cs.colostate.edu/~whitley/Pubs.html]
31. Rego, C. (1996). "Relaxed Tours and Path Ejections for the Traveling Salesman Problems," to appear in the *European Journal of Operational Research*. [http://www.uportu.pt/~crego]
32. Rego, C. and C. Roucairol (1996). "A Parallel Tabu Search Algorithm Using Ejection Chains for the Vehicle Routing Problem," in *Meta-Heuristics: Theory & Applications*, 661-675, I.H. Osman and J.P. Kelly, (eds.), Kluwer Academic Publishers. [http://www.uportu.pt/~crego]
33. Reeves, C.R. (1997). "Genetic Algorithms for the Operations Researcher," *Journal on Computing*, Vol 9, No 3, 231-250 (with commentaries and rejoinder).
34. Rochat, Y. and É. D. Taillard (1995). "Probabilistic diversification and intensification in local search for vehicle routing". *Journal of Heuristics* 1, 147-167. [http://www.idsia.ch/~eric]
35. Taillard, É. D. (1996). "A heuristic column generation method for the heterogeneous VRP", Publication CRT-96-03, Centre de recherche sur les transports, Université de Montréal. To appear in *RAIRO-OR*. [http://www.idsia.ch/~eric]
36. Whitley, D. and J. Kauth, (1988). "GENITOR: A Different Genetic Algorithm," Proceedings of the 1988 Rocky Mountain Conference on Artificial Intelligence.

37. Whitley, D. (1989). The GENITOR Algorithm and Selective Pressure: Why Rank Based Allocation of Reproductive Trials is Best, Morgan Kaufmann, J. D. Schaffer, ed., pp. 116-121.
38. Yamada, T. and C. Reeves (1997). "Permutation Flowshop Scheduling by Genetic Local Search," 2nd IEE/IEEE Int. Conf. on Genetic Algorithms in Engineering Systems (GALESIA '97), pp. 232-238, Glasglow, UK.
39. Yamada, T. and R. Nakano (1996). "Scheduling by Genetic Local Search with Multi-Step Crossover," 4th International Conference on Parallel Problem Solving from Nature, 960-969.

APPENDIX 1 Construction-by-Objective: Mixed Integer and Nonlinear Optimization

The generality of the zero-one diversification generators of Section 3 can be significantly enhanced by a *construction-by-objective* approach, which makes it possible to generate points that satisfy constraints defining a polyhedral region. This approach is particularly relevant for mixed integer programming (MIP) problems and also for nonlinear optimization problems that include linear constraints. Within the MIP context, the resulting solutions can further be processed by standard supporting software to produce associated points that satisfy integer feasibility conditions.

The approach stems from the observation that a specified set of vectors produced by a zero-one diversification generator can instead be generated by introducing appropriately defined objective functions which are optimized over the space of 0-1 solutions. These same objective functions can then be introduced and optimized, either exactly or heuristically, over other solution spaces, to create a more general process for generating trial solutions.

Denote the set of x vectors that satisfy a specified set of constraining conditions by *XC*, and denote the set of zero-one vectors by $X(0,1)$. We then identify a *construction function f(x)* relative to a complementary pair of 0-1 solutions x' and x'' so that

$$x' = argmax(f(x) : x \in X(0,1))$$
$$x'' = argmin(f(x) : x \in X(0,1)).$$

Such a function can be given by any linear function $f(x) = \Sigma(f_j : j \in N)$, where $f_j > 0$ if $x'_j = 1$ and $f_j < 0$ if $x'_j = 0$. Let *H-argmax* and *H-argmin* denote heuristic counterparts of the *argmax* and *argmin* functions, which are based on applying heuristic methods that approximately maximize or minimize *f(x)* over the more complex space *XC*. Then we define solutions *xMax* and *xMin* in *XC* that correspond to x' and x'' in *X(0,1)* by

$$xMax = H\text{-}argmax(f(x) : x \in XC)$$
$$xMin = H\text{-}argmin(f(x) : x \in XC)$$

If *f(x)* is a linear function as specified above, and *XC* corresponds to a bounded feasible linear programming (LP) region, which we denote by *XLP*, then the method that underlies *H-argmax* and *H-argmin* can in fact be an exact method for LP problems, to produce points *xMax* and *xMin* that are linear optima over *XLP*. This observation provides the foundation for a procedure to generate trial solutions for MIP problems.

Following the scatter search strategy of generating weighted centers of selected subregions, we apply the construction-by-objective approach by first identifying a small collection of *primary centers*. These constitute simple centers given by the midpoints of line segments that join pairs of points *xMin* and *xMax*, together with the

center of all such points generated. Additional *secondary centers* are then produced by an accelerated process that does not rely on the exact solution of LP problems, but employs the primary centers to create approximate solutions. Both processes are accompanied by identifying *subcenters*, which are weighted centers of subregions composed by reference to the primary and secondary centers, and include the boundary points *xMax* and *xMin*.

Method to Create Primary Centers

1. Identify a construction function *f(x)* and initialize the set PC of primary centers and the set SubC of subcenters to be empty. Apply a Diversification Generator to produce a small collection of diversified points $x' \in X(0,1)$. For each point generated, execute the following steps.
2. Identify points *xMax* and *xMin* by maximizing and minimizing *f(x)* over *XLP*.
3. Let *xCenter = (xMax + xMin)/2* and add *xCenter* to PC. Also, create associated subcenters *xSubCenter = xMax + w(xMin - xMax)*, where the scalar weight *w* takes the values 0, 1/4, 3/4 and 1, and add these subcenters to the set SubC.
4. After the chosen x' vectors have been generated and processed, create a final point *xCenter** which is the mean of all points *xMax* and *xMin* generated, and add *xCenter** to PC.

The accelerated process for generating secondary centers, which augments the preceding method, continues to rely on a zero-one diversification generator to create 0-1 points $x' \in X(0,1)$, but bypasses LP optimization. The 0-1 points are first mapped into points of an associated region *X(0,U)*, where *U* is a vector of upper bounds chosen to assure that $0 \le x \le U$ is satisfied by all solutions $x \in XLP$. Then each point $x' \in X(0,1)$ produces a corresponding point $xTest \in X(0,1)$ by specifying that $x'_j = 0$ becomes $xTest_j = 0$, and $x'_j = 1$ becomes $xTest_j = U_j$.

Since these created points $xTest \in X(0,U)$ are likely to lie outside XLP, we map them into points of XLP by reference to the elements of PC. For a given *xTest* and a given primary center $xCenter \in$ PC, identify the line

$$L(w) = xCenter + w(xTest - xCenter)$$

where *w* is a scalar parameter. Then a simple calculation identifies the values

$$wMax = Max(w : L(w) \in XLP)$$

$$wMin = Min(w : L(w) \in XLP)$$

which are respectively positive and negative, in the usual condition where *xCenter* does not lie on the boundary of XLP. The associated points of XLP, which are given by

$$xNear = L(wMax)$$
$$xFar = L(wMin),$$

constitute the two points of XLP nearest to and farthest from *xTest* on the line through *xCenter*.

For a given point *xTest*, several primary centers *xCenter* are chosen, thus generating several associated points *xNear* and *xFar*, which are accumulated in sets we denote by *NearSet* and *FarSet*. We then use the function *f(x)* to identify which of the points $xNear \in NearSet$ may take the role of *xMax* and which of the points $xFar \in FarSet$ may take the role of *xMin*. In this case, *xMax* and *xMin* are not determined by solving a linear program, but by identifying the special sets *NearSet* and *FarSet*, and picking the points that qualify as best — i.e, that yield the maximum and minimum values of *f(x)* over these sets of candidates. The result may be viewed as a heuristic approximation to solving the linear programs that maximize and minimize *f(x)* over *XLP*. The process may be described as follows.

Method for Generating Secondary Centers

1. Generate a collection of points $x' \in X(0,1)$ and map each into an associated point $xTest \in X(0,U)$. Begin with the set SC of secondary solutions empty.
2. For each *xTest*, choose one or more elements *xCenter* from PC, including *xCenter**.
3. For each *xCenter* chosen, identify the pair of solutions *xNear* and *xFar* (by reference to *xCenter* and *xTest*), and let *NearSet* and *FarSet* respectively denote the sets that consist of these *xNear* and *xFar* solutions. Then select *xMax* and *xMin* by defining
   $xMax = argmax(f(xNear): xNear \in NearSet)$
   $xMin = argmin(f(xFar): xFar \in FarSet)$
4. Let $xNewCenter = (xMax + xMin)/2$, and add *xNewCenter* to SC. Also, create associated subcenters given by $xSubCenter = xMax + w(xMin - xMax)$, as the scalar weight *w* takes the values 0, 1/4, 3/4 and 1, and add these to SubC.

An alternative to selecting *xMax* and *xMin* by reference to *f(x)* is to choose these solutions to be those that respectively lie closest to and farthest from *xTest*, according to a chosen distance metric *D*, such as given by the L1 or L2 norm. In this case we define

$$xMax = argmin(D(xTest,xNear): xNear \in NearSet)$$
$$xMin = argmax(D(xTest,xFar): xFar \in FarSet)$$

The determination of *xMax* in this way can also be used to map an LP-infeasible point *xTest*, such as one that is generated by a scatter search combination process, into an LP-feasible point *xMax*, similarly using various selected points *xCenter* as a basis for determining associated points *xNear* that are candidates for identifying *xMax*.

It is possible to circumvent the need to identify the vector *U* as a foundation for determining *xTest* by creating *xTest* as a vertex of a unit hypercube that contains *xCenter* as its midpoint, instead of as a vertex of the *(0,U)* region. Specifically, we may define the components of *xTest* to be given by

$$xTest_j = xCenter_j + .5 \;\; if \;\; x_j = 1$$
$$xTest_j = xCenter_j - .5 \;\; if \;\; x_j = 0.$$

Then *xNear* and *xFar* are defined as earlier, so that *xNear* becomes the farthest feasible point from *xCenter* in the positive direction on the ray from *xCenter* through *xTest*, and *xFar* becomes the farthest feasible point from *xCenter* in the negative direction on this ray.

To take advantage of generating larger numbers of points x' by the Diversification Generator, the foregoing process may be modified to create each secondary center from multiple *xTest* points. Specifically, for a chosen number *TargetNumber* of such points, accumulate a vector *xSum* that starts at 0 and is incremented by *xSum = xSum + xMax + xMin* for each pair of solutions *xMax* and *xMin* identified in Step 3 of the foregoing procedure. Once *TargetNumber* of such points have been accumulated, set *xNewCenter = xSum/TargetNumber*, and start again with *xSum* equal to the zero vector. The overall primary center *xCenter** may optionally be updated in a similar fashion.

The values of *w* for generating subcenters can be varied, and used to generate more or fewer points. Also, it is possible to use more than one construction function *f(x)* to create additional points for the preceding approach. For MIP problems, the original objective function can be used to give the general structure for defining a seed solution. The process for transforming the points ultimately selected into solutions that are also integer feasible is examined next.

Creating Reference Solutions.

The primary centers, secondary centers and subcenters provide the source solutions which, after improvement, become candidates for the set RefSet of reference solutions to be combined by scatter search. To create an initial RefSet, we cull out an appropriately dispersed subset of these source solutions by creating a precursor set PreRefSet, whose members will be made integer-feasible and hence become full fledged candidates for RefSet This may be done by starting with the primary center *xCenter** as the first element of PreRefSet, and then applying the criterion indicated in Section 3 to choose each successive element of PreRefSet to be a source solution that maximizes the minimum distance from all solutions thus far added to PreRefSet.

Once PreRefSet reaches its targeted size, the final step is to modify its members by an adaptive rounding process that yields integer values for the discrete variables. For problems small enough that standard MIP software can be expected to generate feasible MIP solutions within a reasonable time, such software can be used to implement the rounding process by introducing an objective function that minimizes the sum of deviations of the integer values from the initial values. (The deviations may be weighted to reflect the MIP objective, or to embody priorities as used in tabu search.) Alternatively, a simple adaptive scheme of the following form can be used to exploit such an objective.

Adaptive Rounding Method to Create Reference Solutions

1. For each integer-infeasible variable in a given candidate solution, introduce a bound at an integer value neighboring its current value, and establish a large LP penalty for deviating from this bound. (Variables not constrained to be integer retain their ordinary objective function coefficients, or can be mildly penalized for deviating from their current values.)
2. Find an LP optimal solution for the current objective. If all integer-constrained variables receive integer values, stop. (The resulting solution is the one sought.)
3. If the LP solution is not MIP feasible, use postoptimality penalty calculations for each integer-constrained variable to identify the cost or profit that results by releasing its current penalty and seeking to drive the variable to a new bound at the closest integer value in the opposite direction from its current bound. Choose the variable that yields the greatest profit (or smallest cost) and impose a penalty for deviating from the indicated new bound. Then return to Step 2.

A natural priority scheme for Step 3 of the preceding approach is to give preference to selecting integer-infeasible variables to be those driven to new bounds, though this type of priority is not as relevant as it is in branch and bound methods. A simple tabu search memory can be used as a basis for avoiding cycling, while allowing greater flexibility of choices than provided by branch and bound.

After applying such a rounding procedure, the transformed members of PreRefSet are ready to be submitted to the customary scatter search processes of applying an Improvement Method and generating combined solutions. (Improvement heuristics can be included as part of the transformation process.) Future diversification phases of the search may be launched at various junctures by following the same pattern as indicated, where the diversification generator may be started from the point where it was discontinued in the preceding phase, or re-started by reference to a different seed solution. The initial PreRefSet for such future phases is populated by elements chosen to maximize the minimum distance from the collection of points previously generated as members of RefSet and PreRefSet, as well as from members currently added to PreRefSet.

APPENDIX 2: Checking for Duplicate Solutions

An additional source of potential duplications arises among solutions x' that are generated as combinations of other solutions (in the phase of Scatter Search or Path Relinking that generates such combined solutions). These solutions x' are inputs to the Improvement Method rather than outputs of this method. By the philosophy of scatter search and path relinking, it is valuable to avoid duplications in these input solutions as well as to avoid duplications in the solutions saved in RefSet. To do this, we store only the r = rNow most recent solutions generated (allowing rNow to grow to a maximum of rMax different solutions recorded), following a scheme reminiscent of a simple short-term recency memory approach in tabu search. In particular, we keep these solutions in an array xsave[r], r = 1 to rNow, and also keep track of a pointer rNext, which indicates where the next solution x' will be recorded once the array is full, i.e., once all rMax locations are filled.

Let E0 and Hash0 be defined for x' as before, and denote associated values for the xsave[r] array by Esave(r) and Hashsave(r). These are accompanied by a "depth" value, which is 0 if no duplication occurs, and otherwise tells how deep in the list – how far back from the last solution recorded – a duplication has been found. For example, depth = 3 indicates that the current solution duplicates a solution that was recorded 3 iterations ago. (This is not entirely accurate, since, for example, depth = 3 could mean the solution was recorded 5 iterations ago and then 2 other duplications occurred, which still results in recording only 3 solutions.)

An appropriate value for rMax can be determined by initial testing that sets this value larger than expected to be useful. An array CountDup(depth), for depth = 1 to rMax, can then be kept that counts how often duplications are found at various depths. If the array discloses thatvery few duplications occur for depths beyond a given value, then rMax can be reduced to such a value, without the risk of having to process many solutions that duplicate others encountered. (Although the reduced value of rMax will save some effort checking for duplications, it may be the effort will not be too great anyway, if a quick check based on using Hash0 can screen out most of the potential duplications.)

To keep track of auxiliary information we introduce counters corresponding to DupCheck, FullDupCheck and FullDupFound of the RefSet Update Routine, which we give the names DupCheckA, FullDupCheckA, and FullDupFoundA. Finally, we keep track of the number of times the routine is called by a value DupCheckCall.

Initialization Step:

rNow = 0
rNext = 0
CountDup(depth) = 0, for depth = 1 to rMax
DupCheckA = 0
FullDupCheckA = 0

FullDupFoundA = 0
DupCheckCall = 0

Duplication Check Subroutine.
Begin Subroutine.
DupCheckCall = DupCheckCall + 1
depth = 0
If rNow = 0 then:
rNow = 1; rNext = 1;
xsave[1] = x' (record x' in xsave[1]),
Esave(1) = E0; Firstsave(1) = FirstIndex0
End the Subroutine
Elseif rNow > 0 then:
(Go through the solutions in "depth order", from the one most recently stored to the one least recently stored. When a duplication is found, the loop index r (below) indicates the value of rMax that would have been large enough to identify the duplication.)
i = rNext
For r = 1 to rNow
If Esave(i) = E0 then:
DupCheckA = DupCheckA + 1
If Hash0 = Hashsave(i) then:
FullDupCheckA = FullDupCheckA + 1
If x' = x[i] then:
(x' duplicates a previous solution)
FullDupFoundA = FullDupFoundA + 1
depth = r
CountDup(depth) = CountDup(depth) + 1
End the Duplication Check Subroutine
Endif
Endif
Endif
i = i-1
if i < 1 then i = rNow
End r
(Here, no solutions were duplicated by x'. Add x' to the list in position rNext, which will replace the solution previously in rNext if the list is full.)
rNext = rNext + 1
If rNext > rMax then rNext = 1
If rNow < rMax then rNow = rNow + 1
xsave[rNext] = x'
Esave(rNext) = E0
Hashsave(rNext) = Hash0
Endif
End of Duplication Check Subroutine