An Experimental Evaluation of a Scatter Search for the Linear Ordering Problem

Vicente Campos[a], Fred Glover[b], Manuel Laguna[b], and Rafael Martí[a]

a Dpto. de Estadística e Investigación Operativa, Facultad de Matemáticas, Universitat de Valencia, Dr. Moliner, 50, 46100 Burjassot, Valencia, Spain. Vicente.Campos@uv.es and Rafael.Marti@uv.es

b Graduate School of Business and Administration, Campus Box 419, University of Colorado, Boulder, CO 80309-0419, USA. Fred.Glover@Colorado.edu and Manuel.Laguna@Colorado.edu

Latest Revision: April 23, 1999

Abstract — Scatter search is a population-based method that has 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, such as the surrogate constraint method, scatter search uses strategies for combining solution vectors that have proved effective in a variety of problem settings.

In this paper, we present a scatter search implementation designed to find high quality solutions for the NP-hard linear ordering problem, which has a significant number of applications in practice. The LOP, for example, is equivalent to the so-called triangulation problem for input-output tables in economics. Our implementation goes beyond a simple exercise on applying scatter search, by incorporating innovative mechanisms to combine solutions and to create a balance between quality and diversification in the reference set. We also use a tracking process that generates solution statistics disclosing the nature of combinations and the ranks of antecedent solutions that produced the best final solutions. Our extensive computational experiments with more than 300 instances establishes the effectiveness of our procedure in relation to those previously identified to be best.

1. Introduction

The goal of the research presented in this paper is to expand the scatter search methodology by implementing innovative strategies to apply its underlying framework to the linear ordering problem. The linear ordering problem (or LOP) has generated a considerable amount of research interest over the years, as documented in Grotschel, et al. (1984) and Chanas and Kobylanski (1996). Because of its practical and theoretical relevance, we use this problem as a test case for our strategies and search mechanisms.

Given a matrix of weights $E = \{ e_{ij} \}_{m \times m}$, the LOP consists of finding a permutation p of the columns (and rows) in order to maximize the sum of the weights in the upper triangle. In mathematical terms, we seek to maximize:

$$C_E(p) = \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} e_{p(i)p(j)} .$$

where $p(i)$ is the index of the column (and row) in position i in the permutation. Note that in the LOP, the permutation p provides the ordering of both the columns and the rows. The equivalent problem in graphs consists of finding, in a complete weighted graph, an acyclic tournament with a maximal sum of arc weights (Reinelt, 1985).

In economics, the LOP is equivalent to the so-called *triangulation problem for input-output tables*, which can be described as follows. The economy of a region (generally a country) is divided into m sectors and an $m \times m$ input-output table E is constructed where the entry e_{ij} denotes the amount of deliveries (in monetary value) from sector i to sector j in a given year. The triangulation problem then consists of simultaneously permuting the rows and columns of E, to make the sum of the entries above the main diagonal as large as possible. An optimal solution then orders the sectors in such a way that the suppliers (i.e., sectors that tend to produce materials for other industries) come first, followed by the consumers (i.e., sectors that tend to be final-product industries that deliver their output mostly to end users).

Instances of input-output tables from sectors in the European Economy can be found in the public-domain library LOLIB (1997). In the computational experimental section, we employ these problem instances to test the merit of our scatter search implementation.

Scatter search, from the standpoint of metaheuristic classification, may be viewed as an evolutionary (population-based) algorithm that constructs solutions by combining others. It derives its foundations from strategies originally proposed for combining decision rules and constraints in the context of integer programming. The goal of this methodology is to enable the implementation of solution procedures that can derive new solutions from combined elements in order to yield better solutions than those procedures that base their combinations only on a set of original elements. E.g., see the overview by Glover (1998).

In the area of scheduling, researchers introduced the notion of combining rules to obtain improved local decisions. Numerically weighted combinations of existing rules, suitably restructured so that their evaluations embodied a common metric, generated new rules. The approach was motivated by the conjecture that information about the relative desirability of alternative choices is captured in different forms by different rules, and that this information can be exploited more effectively when integrated than when treated in isolation (i.e., by choosing selection rules one at a time). Empirical outcomes disclosed that the decision rules created from such combination strategies

produced better outcomes than standard applications of local decision rules. The strategy of creating combined rules also proved superior to a "probabilistic learning approach" that used stochastic selection of rules at different junctures, but without the integration effect provided by the combined rules (Crowston, et al., 1963; Fisher and Thompson, 1963).

The associated procedures for combining constraints likewise employed a mechanism of generating weighted combinations. In this case, nonnegative weights were introduced to create new constraint inequalities, called *surrogate constraints*, in the context of integer and nonlinear programming (Glover, 1965, 1968). The approach isolated subsets of (original) constraints that were gauged to be most critical, relative to trial solutions that were obtained based on the surrogate constraints. This critical subset was used to produce new weights that reflected the degree to which the component constraints were satisfied or violated. In addition, the resulting surrogate constraints served as source constraints for deriving new inequalities (cutting planes) which in turn provide material for creating further surrogate constraints.

The main function of surrogate constraints was to provide ways to evaluate choices that could be used to generate and modify trial solutions. A variety of heuristic processes that employed such evaluations evolved from this foundation. As a natural extension, these processes led to the related strategy of combining solutions, as a primal counterpart to the dual strategy of combining constraints, which became manifest in scatter search.

2. Main Scatter Search Elements

Our solution method for the linear ordering problem is based on the scatter search template in Glover (1998). The procedure combines the following elements:

a) Diversification Generator
b) Improvement Method
c) Reference Set Update Method
d) Subset Generation Method
e) Solution Combination Method

We describe the implementation of each of these elements, as adapted in the context of the linear ordering problem.

a) Diversification Generator

This element of the scatter search approach is particularly important, given the goal of developing a method that balances diversification and intensification in the search. For this purpose, we have developed and tested 10 diversification generation methods. Six of these methods are based on GRASP (Feo and Resende, 1995) constructions with a greedy function that selects sectors based on a measure of attractiveness.

DG01. A GRASP construction where the attractiveness of a sector is the sum of the elements in its corresponding row. The method randomly selects from a short list of the most attractive sectors and constructs the solution starting from the first position of the permutation.

DG02. A GRASP construction where the attractiveness of a sector is the sum of the elements in its corresponding column. The method randomly selects from a short list of the most attractive sectors and constructs the solution starting from the first position of the permutation.

DG03. A GRASP construction where the attractiveness of a sector is the sum of the elements in its corresponding row divided by the sum of the elements in its corresponding column. The method randomly selects from a short list of the most attractive sectors and constructs the solution starting from the first position of the permutation.

DG04, DG05 and DG06. These methods are identical to the first three, except that the selection of sectors is from a short list of the least attractive and the solution is constructed starting from the last position of the permutation.

DG07. A mixed procedure derived from the previous 6. The procedure generates an even number of solutions from each of the previous six methods and combines these solutions in a single set. That is, if n solutions are required, then each method DG0i (for i = 1, …, 6) contributes with $n/6$ solutions.

DG08. A random generator. This method simply generates random permutations.

DG09. A method suggested in Glover (1998) which generates diversified permutations in a systematic way without reference to the objective function.

DG10. A method using frequency-based memory, as proposed in tabu search (Glover and Laguna, 1997). This method is based on modifying a measure of attractiveness proposed by Becker (1967) with a frequency measure that discourages sectors from occupying positions that they have frequently occupied in previous solution generations.

We now provide a detailed description of DG10, and then show that this generator outperforms the others in terms of solution quality and diversification power. The DG10 generator is based on the notion of constructing solutions employing modified frequencies. The generator exploits the permutation nature of a linear ordering. A frequency counter is maintained to record the number of times an element i appears in position j. The frequency counters are used to penalize the "attractiveness" of an element with respect to a given position, as in the approach of Laguna and Glover (1993). To illustrate, suppose that the generator has created 30 solutions to be included in a set of solutions P. If 20 of the 30 solutions have element 3 in position 5, then the frequency counter $freq(3,5) = 20$. This frequency value is used to bias the potential assignment of element 3 in position 5 during subsequent constructions, and therefore, inducing diversification with respect to the solutions already in P.

The attractiveness of assigning element i to position j is given by the greedy function $G(i,j)$, as proposed in Becker (1967). We modify the value of $G(i,j)$ to reflect previous assignments of element i to position j, as follows:

$$G'(i,j) = G(i,j) - \beta * \left(\frac{MaxG}{MaxF} \right) * Freq(i,j).$$

Where $MaxF$ is the maximum $Freq(i,j)$ value for all i and j, and $MaxG$ is the maximum $G(i,j)$ value for all i and j.

It is important to point out that in each iteration of the construction procedure, $G(i,j)$ is an adaptive function since its value depends on attributes of the unassigned elements. A pseudo-code of the diversification generator to produce a set P with $PSize$ solutions, appears in Figure 1.

Figure 1. Pseudo-code of diversification generator DG10.

1. $P = \emptyset$.
2. $Freq(i,j) = 0$, for all i and j.

While ($|P| < PSize$) **do**

3. $p(i) = 0$ for all i.
4. $N = \{ 1,\ldots, m \}$ (The set of unassigned elements.)
5. $j = MaxF = MaxG = 1$.

While ($N \neq \emptyset$) **do**

6. $G'(i,j) = G(i,j) - \beta * \left(\frac{MaxG}{MaxF}\right) * Freq(i,j) \qquad \forall i \in N$.
7. $i^* = \arg\max_i (G(i,j), i \in N)$
8. $MaxG = G(i^*,j)$
9. $N = N - \{ i^* \}$
10. $p(i^*) = j$
11. $j = j + 1$

End while

12. $Freq(i,p(i)) = Freq(i,p(i)) + 1$, for all i.
13. $MaxF = \max_{i,j} (Freq(i,j), i \in N, j \in \{1,\ldots,m\})$

If ($p \notin P$) then $P = P \cup \{p\}$

End while

The pseudo-code in Figure 1 is written using general mathematical notation. However, the actual implementation takes advantage of quick updating mechanisms that are hard to represent mathematically. For example, the value of $MaxF$ in step 13 can be maintained by keeping track of the updates of the frequency matrix $Freq$.

Clearly, the performance of the diversification generator depends on the value of the parameter β. In order to determine effective values for this parameter, we have created a measure of diversity for a set of solutions. The diversity measure is calculated as follows:

1. Calculate the median position of each sector i in the solutions in P.
2. Calculate the dissimilarity of each solution in the population with respect to the median position. The dissimilarity is calculated as the sum of the absolute difference between the position in the solution under consideration and the median solution.
3. Calculate d as the sum of all the individual dissimilarities.

To illustrate, suppose that P consists of the following three orderings: (A,B,C,D), (B,D,C,A), (C,B,A,D). Then the median position of sector A is 3, since it occupies positions 1, 3 and 4 in the given orderings. In the same way, the median positions of B, C and D are 2, 3 and 4, respectively. Note that the median positions might not induce an ordering, as in the case of this example. The dissimilarity of the first solution is then calculated as follows:

$$d_1 = |1-3| + |2-2| + |3-3| + |4-4| = 2.$$

In the same way, the dissimilarities of the other two solutions are $d_2 = 4$ and $d_3 = 2$. The diversity measure for P is then given by $d = 2+4+2 = 8$.

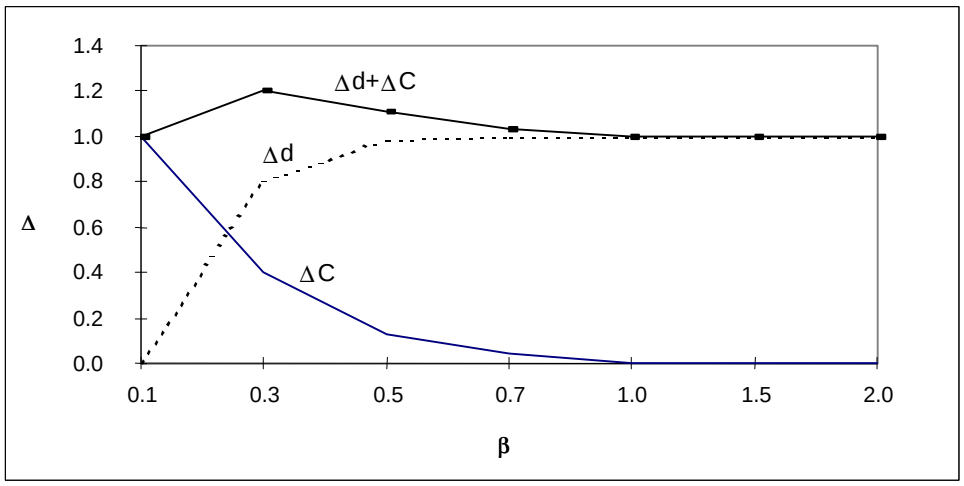
Since we are interested in finding a β-value that generates balanced populations, we use two relative measures: one based on the diversity value d, and one based on the objective function value C. Given a set of populations $\Pi = \{ P_1, P_2, \ldots, P_n \}$, where P_i has been generated with β_i, we calculate $\Delta d(i)$ and $\Delta C(i)$ for each population P_i as follows:

$$\Delta d(i) = \frac{d(i) - d_{\min}}{d_{\max} - d_{\min}}, \text{ where } d(i) = \sum_{j \in P_i} d_j, \; d_{\min} = \min_i \{d(i)\} \text{ and } d_{\max} = \max_i \{d(i)\};$$

$$\Delta C(i) = \frac{C(i) - C_{\min}}{C_{\max} - C_{\min}}, \text{ where } C(i) = \sum_{j \in P_i} C_j, \; C_{\min} = \min_i \{C(i)\} \text{ and } C_{\max} = \max_i \{C(i)\}.$$

Figure 2 shows the values of our relative measures of quality and diversity when the diversification generator is executed with different β-values.

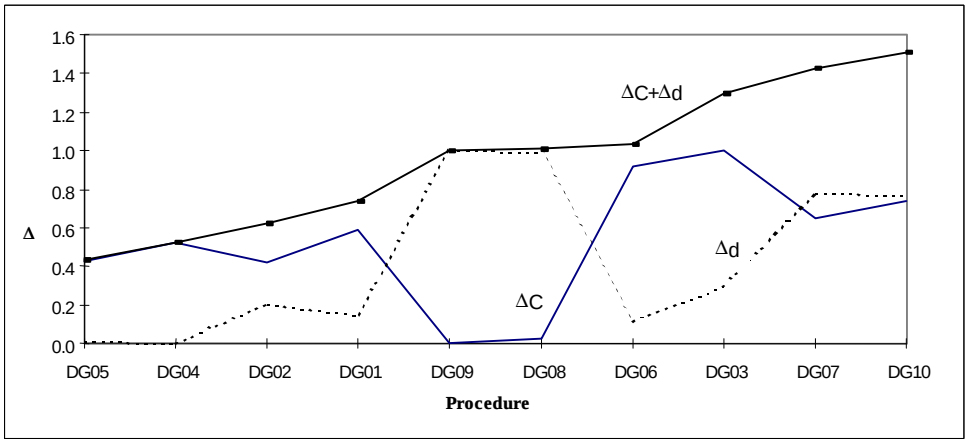
Figure 2. Diversity vs. quality for different β-values.



This figure discloses that a value of 0.3 for β provides a good balance between diversity and quality in the population. As β increases, the diversity increases but the quality decreases. Therefore the value of $\Delta d(i) + \Delta C(i)$ is highest at $\beta = 0.3$, with a similar contribution from both relative deviation measures.

We use our measure of diversity and quality to compare the performance of the 10 competing diversification generators. Figure 3 shows the ΔC, Δd and $\Delta C + \Delta d$ values achieved by each procedure.

Figure 3. Diversity vs. quality for alternative Diversification Generation Methods.



As expected, the random generator (DG08) produces the maximum diversity (as measured by the dissimilarity value). DG09 matches the diversity of DG08 using a systematic approach instead of randomness. The mixed method DG07 provides a good balance between dissimilarity and quality, by the union of solutions generated with methods DG01 to DG06. Clearly, DG10 outperforms the competing methods, by producing the highest combined score.

b) Improvement Method

The improvement method is based on the neighborhood search developed for the LOP in Laguna, Martí and Campos (1998). Two neighborhoods were considered in that study:

$N_1 = \{p' : \text{INSERT\_MOVE}(p(j), i), \text{ for } j = 1, \ldots, m\text{-}1 \text{ and } i = j\text{+}1\}$
$N_2 = \{p' : \text{INSERT\_MOVE}(p(j), i), \text{ for } j = 1, \ldots, m \text{ and } i = 1, 2, \ldots, j\text{-}1, j\text{+}1, \ldots, m\}$

N_1 consists of permutations that are reached by switching the positions of contiguous sectors $p(j)$ and $p(j+1)$. N_2 consists of all permutations resulting from executing general insertion moves, as defined above. In conjunction with these neighborhoods, two strategies are defined. The *best* strategy selects the move with the largest move value among all the moves in the neighborhood. The *first* strategy, on the other hand, scans the list of sectors (in the order given by the current permutation) in search for the first sector $p(f)$ whose movement results in an strictly positive move value (i.e., a move such that $C_E(p') > C_E(p)$). The move selected by the *first* strategy is then INSERT_MOVE($p(f)$, i^*), where i^* is the position that maximizes $C_E(p')$. Note that for N_1, $i^* = f+1$, while for N_2, i^* is chosen from $i = 1, 2, \ldots, f\text{-}1, f\text{+}1, \ldots, m$. Therefore, the *first* strategy used in combination with N_1 is equivalent to searching for the first improving move in the neighborhood.

Combining the selection strategies with the neighborhood definitions results in four greedy local search procedures: *first*(N_1), *best*(N_1), *first*(N_2), and *best*(N_2). The experimentation in Laguna, Martí and Campos (1998) showed that the greedy procedure *first*(N_2) was the most effective. Based on this finding, we partition N_2 into m N_2^j neighborhoods

$$N_2^j = \{p' : \text{INSERT\_MOVE}(p(j), i),\ i = 1, 2, \ldots, j\text{-}1, j\text{+}1, \ldots, m\}$$

associated with each sector $p(j)$, for $j = 1, \ldots, m$. We therefore base our improvement method on choosing the best insertion associated with a given sector as proposed in Laguna, Martí and Campos (1998).

c) Reference Set Update Method

The Reference Set Update Method accompanies each application of the Improvement Method, and is generally examined right after the Improvement Method, because of its linking role with the Subset Generation Method. The update operation consists of maintaining a record of the b best solutions found, where the value of b is treated as a constant search parameter. The issues associated with this updating function are conceptually straightforward.

Glover (1998) provides a detailed pseudo-code description of the procedure for maintaining *RefSet* (i.e., the Reference Set), called the *RefSet* Update Routine. This routine is organized to handle vectors of 0-1 variables. We have adapted the *RefSet* Update Routine to handle permutation vectors, as needed in our current application.

d) Subset Generation Method

This procedure consists of creating different subsets X of *RefSet*, as a basis for implementing the subsequent combination method. The scatter search methodology prescribes that the set of combined solutions (i.e., the set of all combined solutions that the implementation intends 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, where the combinations are typically determined by the spin of a roulette wheel.

The procedure seeks to generate subsets X of *RefSet* that have useful properties, while avoiding the duplication of subsets previously generated. The approach for doing this is organized to generate four different collections of subsets of *RefSet*, *SubSetType* 1, 2, 3 and 4, with the following characteristics:

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 b.

A central consideration of this design is that *RefSet* itself might not be static, because it might 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). In our implementation, however, we maintain a static updating of *RefSet*, but use a broad definition of “best” for the membership in this set. In other words, we do not allow *RefSet* to dynamically change its size, but we use two criteria to allow solutions to initially become members of this set. One criterion is the quality of the solution (as given by the objective function value) and the other is the diversity of the solution (as

given by the dissimilarity measure). In this sense, our definition of "best" to construct the first *RefSet* is broader than one that considers only the value of the objective function.

After the first *RefSet* has been created, subsequent membership in the set can only be obtained by means of solution quality. That is, *RefSet* changes when the Combination Method generates solutions of higher quality and the process stops when *RefSet* converges.

e) Solution Combination Method

The Solution Combination Method as well as the Improvement Method are elements of scatter search that are context-dependent. Although it is possible to design "generic" combination procedures, it is more effective to base the design on specific characteristics of the problem setting. Our Solution Combination Method, which is applied to each subset generated in the previous step, uses a min-max construction based on votes.

The method scans (from left to right) each reference permutation in the subset, and uses the rule that each reference permutation votes for its first element that is still not included in the combined permutation (referred to as the "incipient element"). The voting determines the next element to enter the first still unassigned position of the combined permutation. This is a min-max rule in the sense that if any element of the reference permutation is chosen other than the incipient element, then it would increase the deviation between the reference and the combined permutations. Similarly, if the incipient element were placed later in the combined permutation than its next available position, this deviation would also increase. So the rule attempts to minimize the maximum deviation of the combined solution from the reference solution, subject to the fact that other reference solutions in the subset are also competing to contribute.

This voting scheme can be implemented using a couple of variations that depend on the way votes are modified:

(V1) The vote of a given reference solution is weighted according to the incipient element's position (referred to as the "incipient position"). A smaller incipient position gets a higher vote. For example, if the element in the first position of some reference permutation is not assigned to the combined permutation during the first 4 assignments, then the vote is weighted more heavily to increase the chances of having that element assigned to position 5 of the combined permutation. The rule emphasizes the preference to this assignment versus having a later occurring element of some other reference permutation (which is the incipient element for that other permutation) become assigned.

(V2) A bias factor that gives more weight to the vote of a reference permutation with higher quality. Within the current organization of our scatter search implementation such a factor should be very slight because it is expected that high quality solutions will be strongly represented anyway.

We chose to implement V1 with a tie-breaking rule that is based on solution quality. The rule is used when more than one element receives the same votes. Then the element with highest weighted vote is selected, where the weight of a vote is directly proportional to the objective function value of the corresponding reference solution.

3. Overall Procedure

The proposed procedure can be summarized as follows:

1. Start with $P = \emptyset$. Use the diversification generator DG10 to construct a solution s. Apply the Improvement Method to s to obtain the improved solution s^*. If $s^* \notin P$ then, add s^* to P (i.e., $P = P \cup s^*$), otherwise, discard s^*. Repeat this step until $|P| = PSize$.

2. Order the solutions in P according to their objective function value (where the best overall solution is first on the list).

3. Build *RefSet* from P, with $|RefSet| = b_1+b_2$ (i.e., $b = b_1+b_2$). Take the first b_1 solutions in P and assign them to *RefSet*. For each solution x in *P-RefSet*, calculate the minimum dissimilarity $d(RefSet,x)$ to all solutions in *RefSet*. Select the solution x' with the maximum dissimilarity $d(RefSet,x')$ of all x in *P-RefSet*. Add x' to *RefSet*, until $|RefSet| = b_1+b_2$. Make *NewElements* = TRUE.

While (*NewElements*) **do**

4. Calculate the number of subsets (*MaxSubset*) that include at least one new element. Make *NewElements* = FALSE.

For (*SubsetCounter* = 1, ..., *MaxSubset*) **do**

5. Generate the next subset r from *RefSet* with the Subset Generation Method. This method generates one of four types of subsets with number of elements ranging from 2 to $|RefSet|$. Let subset $r = \{ r_1, ..., r_k \}$, for $2 \le k \le |RefSet|$.

6. Apply the Solution Combination Method to r to obtain a new solution s_r.

7. Apply the Improvement Method to s_r, to obtain the improved solution s_r^*.

If (s_r^* is not in *RefSet* and the objective function value of s_r^* is better than the objective function value of the worst element in *RefSet*) **then**

8. Add s_r^* to *RefSet* and delete the worst element currently in *RefSet*.
9. Make *NewElements* = TRUE.

End if

End for

End while

The procedure starts with the generation of *PSize* distinct solutions. These solutions are originally generated to be diverse and subsequently improved by the application of the Improvement Method (step 1). The set P of *PSize* solutions is ordered in step 2, in order to facilitate the task of creating the reference set in step 3. The reference set (*RefSet*) is constructed with the first b_1 solutions in P and b_2 solutions that are diverse with respect to the members in *RefSet*.

The search consists of a "while-loop" and a "for-loop" that are designed to control both the presence of new elements and the examination of all the subsets with at least one new element. In step 4, the number of subsets with at least one new element is counted and this value is assigned to *MaxSubset*. The Boolean variable *NewElements* is made FALSE before the subsets are examined, since it is not known whether a new solution will enter the reference set in the current examination of the subsets. The actual generation of the subsets occurs in step 5. Note that only subsets with at least one new element are generated in this step. A solution is generated in step 6 by applying the Combination Method. Step 7 attempts to improve this solution with the application of the Improvement Method. If the improved solution from step 7 is better than the worst solution in the reference set, then the improved solution becomes a new element of *RefSet*. The solution is added in step 8 and the *NewElements* indicator is switched to TRUE in step 9.

We now turn our attention to the computational experiments used to assess the merit of our scatter search implementations. Details of such experiments are provided in the next section.

4. Computational Experiments

The overall procedure described in section 3 was implemented in C, and all experiments were performed on a Pentium 166 Mhz personal computer.

Parameter Tuning

The first set of experiments is designed to find the best values for the key parameters of our scatter search implementation. For this experiment, we use 15 instances from the public-domain library LOLIB (1997) and 15 randomly generated instances with 100 sectors and with weight values uniformly distributed between 0 and 100. The following values were tested during these experiments:

PSize	50, 100, and 150
b	10, 20 and 40
(b_1, b_2)	(5, 5), (10,10), (5, 15), (15, 5) and (20, 20)

The experiments reveled that a significant change in the solution quality is due to the increase in *PSize*. The experiments were inconclusive about the advantage of increasing the size of the reference set (i.e., the value of *b*) beyond 20 when *PSize* is no more than 100. In the same way, the experiments showed that the best results are obtained when $b_1 = b_2$. For the next set of experiments, we set our key parameters to the following values: *PSize* = 100, *b* = 20 and $b_1 = b_2 = 10$.

Tracking Combination Strategies

This experiment is designed to assess the contribution of the different types of combinations embedded in our implementation. In other words, we would like to know if the best solution was generated from the combination of 2, 3 or 4 reference solutions. (The answer could be "all of the above", since the best solution can result from a succession of combinations, each of a different type.) In general, we undertake to identify how often, across a set of benchmark problems, the best solutions came from various combinations of *k* reference solutions.

Since *SubsetType* 1 through 4 respectively generate solutions from 2 to up to *b* reference solutions, we keep a 4 element array RS(*i*) for each solution in rank *i* of *RefSet* (corresponding to *SubsetType* 1 to 4). The array starts (0,0,0,0), meaning that there are

no sources, and then counts the number of times those different subset types are used. For instance, suppose three solutions with (2,0,0,1), (5,1,0,0) and (0,1,0,0) are combined by *SubsetType* 2. Then the new solution is accompanied by the array (7,3,0,1) — the sum of the other arrays, plus 1 added to position 2.

We use 15 instances from LOLIB and the RS arrays corresponding to the solutions generated during the search to find the percentage of time that each subset type is used to combine solutions that become members of the reference set. The percentages are 86.9%, 11%, 2% and 0.1%, for *SubsetType* 1, 2, 3 and 4, respectively. This experiment provides an idea of the relative importance of these combination strategies. (However, we observe that these percentages could change if the subset types were generated in a different sequence.)

We perform the same experiment with 15 randomly generated instances with 100 sectors and with weight values uniformly distributed between 0 and 100. The results are similar to those obtained before, with values of 80.3%, 15.1%, 4.3% and 0.4%, for *SubsetType* 1, 2, 3, and 4, respectively.

Tracking Best Solutions

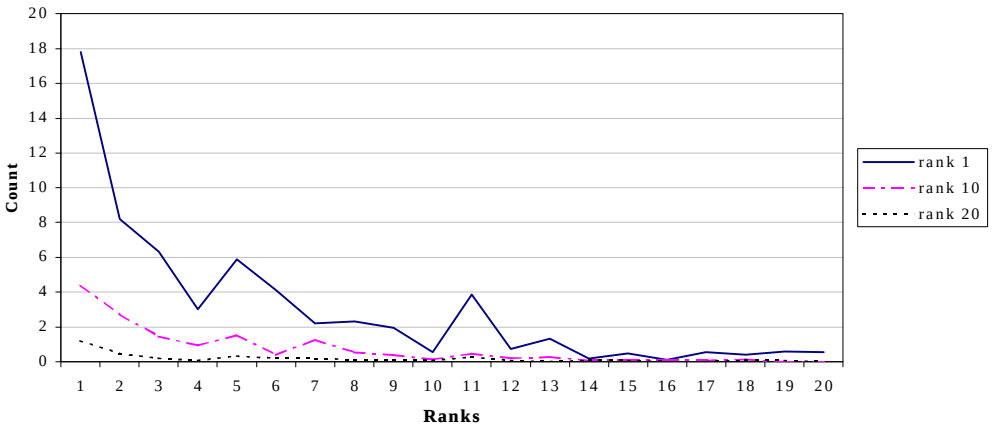
An item of special relevance in scatter search is to know not only how often the best solutions come from different types of combinations, but also to know the ranks of the reference solutions that generated these best solutions. That is, a trace could be enlightening that says: the overall best solution came from combining the 3rd and 5th best solutions — where one of these came from combining the 1st, 2nd and 6th best solutions, and the other came from ... etc. This trace gives us an idea of what solutions are important as components of others.

For this experiment, we keep at most 20 best solutions in *RefSet*. Then we create a 20 by 20 matrix *Source*(*i*,*j*), where *Source*(*i*,*j*) counts the number of times a solution of rank *j* was a reference point for the current solution for rank *i*. (To begin, this matrix is "all 0".) Then, suppose next that the current solution to be assigned rank 2 is created from 3 reference points, of rank 1, 4 and 6. Before shifting the records so that the previous rank 2 will be the new rank 3, etc., we create a "working space" *Work*(*j*) that will become the row *Source*(2,*j*) after the shift. *Work*(*j*) in the present example just sums the contents of the three rows *Source*(1,*j*), *Source*(4,*j*) and *Source*(6,*j*), and then also increases *Work*(1), *Work*(4) and *Work*(6) by 1. The data structure is maintained and updated by this basic type of process.

Figure 4 shows the average of rows 1, 10 and 20 of the *Source* matrix after running scatter search on 15 LOLIB instances.

The interpretation of the lines in Figure 4 is as follows. Consider the line associated with rank 1. Then, a count of almost 18 in the first point of this line indicates that rank 1 solutions were generated 18 times from other rank 1 solutions. Similarly, rank 1 solutions were generated 8 times from rank 2 solutions. The decaying effect exhibited by all the lines indicate that high quality solutions tend to generate new solutions that are admitted to the reference set. This is evident by the counts corresponding to rank 1 in the x-axis. During the search, rank 1 solutions generated 18 rank 1 solutions, 4 rank 10 solutions and 1 rank 20 solution, on average.

Figure 4. Plot of rows 1, 10 and 20 from the *Source* matrix.



We performed the same experiment using randomly generated instances and observed the same behavior depicted in Figure 4. The magnitude of the counts, however, drastically changed, because the number of solutions added to the reference set in the LOLIB problems is in the order of 20, while this value is in the order of 100 for randomly generated problems.

Assessing Performance

In the following set of experiments we compare the performance of two variants of our scatter search implementation with three methods: Chanas and Kobylanski (1996), Tabu Search (Laguna, Martí and Campos, 1998) and a greedy procedure described below. The two variants of scatter search are SS20-100 and SS40-200, indicating that b = 20 and 40 and *PSize* = 100 and 200, respectively. In both variants, $b_1 = b_2$. Given the results of our experiments tracking the combination method, we restrict the procedure to exclusively operate with *SubsetType* 1, 2 and 3.

The greedy local search procedure is based on deleting a sector from its current position and inserting it in another position. An iteration of the algorithm consists of scanning the list of sectors in search for the first one whose movement results in an improvement of the objective function value. The algorithm ends when there is no improving move.

We have tested the procedures on four sets of instances:

(1) *LOLIB Instances*. These instances from the public-domain library consist of input-output tables from sectors in the European economy. Total number of instances is 49.

(2) *SGB Instances*. These instances from the Stanford GraphBase (Knuth, 1993) consist of input-output tables from sectors in the economy of the United States. The set has a total of 25 instances with 75 sectors.

(3) *Random Type I.* These instances are generated from a (0,100) uniform distribution. Reinelt (1985) considers these instances. We generate

instances of sizes ranging from 35 to 200. There are 25 instances in each set for a total of 125.

(4) *Random Type II.* These instances are generating by counting the number of times a sector appears in a higher position than anther in a set of randomly generated permutations. For a problem of size m, $m/2$ permutations are generated. Chanas and Kobilansky (1996) consider these instances. We generate instances of sizes 100, 150 and 200. There are 25 instances in each set for a total of 75.

To reduce the computational effort related to avoiding duplications in the population, we have implemented the following hash function to compare solutions and avoid duplications:

$$hash(p) = \sum_{i=1}^{m} ip(i)^2$$

We have empirically determined that two different solutions almost always have different hash values (with a few rare exceptions). Therefore, in our implementation, when two solutions have the same hash value we consider that both are the same and eliminate one of them from further consideration. This implementation results in average computational-time savings of about 7% in LOLIB instances and 1.5% in random instances when compared to a full duplication checking mechanism. The scatter search procedure yields significantly lower duplications in random instances than in real-world examples. Intuitively, this can be explained by the relationships between sectors in real-world examples, which tend to favor the generation of similar solutions.

Tables 1 to 5 show, for each procedure, the average objective function value, the average percent deviation from optimality, the number of optimal solutions, and the average CPU time in seconds. Since optimal solutions are not known for the SGB and the large random instances, the deviation in Tables 3, 4 and 5 is reported considering the best solution found during each experiment. Also for these tables, the number of best solutions found is reported instead of the number of optimal solutions. We refer to Chanas and Kobylanski's method as CK, and as CK10 to the application of the method from 10 randomly generated initial solutions. The tabu search method will be denoted as TS.

Table 1. Results with LOLIB instances.

	greedy	**CK**	**CK10**	**TS**	**SS20-100**	**SS40-200**
value	22033729.5	22018008.3	22040892.1	22040108.5	22041229.8	22041234.9
optima dev.	0.15%	0.15%	0.02%	0.04%	0.01%	0.01%
number of optima	11	11	27	33	42	44
run time (sec.)	0.01	0.10	1.06	0.49	2.35	7.1

Table 2. Results with random type I instances (m = 35).

	greedy	**CK**	**CK10**	**TS**	**SS20-100**	**SS40-200**
value	34325.6	34293.0	34444.5	34463.3	34487.1	34487.1
optima dev.	0.47%	0.56%	0.12%	0.07%	0.00%	0.00%
number of optima	0	1	4	16	25	25
run time (sec.)	0.00	0.03	0.26	0.22	2.21	8.65

Table 3. Results with SGB Instances.

	greedy	CK	CK10	TS	SS20-100	SS40-200
value	6125873.0	6141798.8	6144422.7	6144287.5	6144980.8	6145072.3
best dev.	0.312%	0.054%	0.011%	0.013%	0.002%	0.001%
number of best	0	0	0	5	18	22
run time (sec.)	0.10	2.73	31.93	3.03	14.05	51.81

Table 4. Results with random type I instances (m = 100, 150 and 200).

	greedy	CK	CK10	TS	SS20-100	SS40-200
value	645919.8	645809.3	646790.6	648030.4	648591.3	648705.4
best dev.	0.45%	0.48%	0.29%	0.11%	0.02%	0.01%
number of best	0	0	0	5	21	49
run time (sec.)	0.08	6.90	67.12	12.74	63.82	217.27

Table 5. Results with random type II instances (m = 100, 150 and 200).

	greedy	CK	CK10	TS	SS20-100	SS40-200
value	549997.0	550010.3	550067.3	550114.3	550112.3	550118.1
best dev.	0.02%	0.02%	0.01%	0.00%	0.00%	0.00%
number of best	0	0	0	41	14	45
run time (sec.)	0.07	4.30	44.51	7.91	43.36	170.93

The greedy procedure is clearly inferior in terms of solution quality, although given its simplicity, its performance is quite acceptable. The performance of the greedy and CK methods is very similar within each of the four problem sets, but their deviation from the optimal (or best) solutions is significantly higher in the case of the random instances type I. Both the greedy and the CK procedures are performed from a randomly generated initial solution. TS and the SS variants, on the other hand, are quite robust, as evident by the negligible change in the deviation values across tables.

It is difficult to measure solution quality in terms of percent deviation, since TS and both SS variants have very small average deviations from optimality. In terms of number of optima (or best solutions), TS is very competitive, considering that is able to match 33 (out of 49) optima in the LOLIB and 16 (out of 25) optima in the small random type I problems (see Tables 1 and 2). TS is also computationally efficient, requiring a maximum of less than 13 seconds per problem, on the average. The most robust method is SS40-200 in terms of number of optima or best solutions found. However, this is done at the expense of higher computational times.

It should be mentioned that the optimal solutions to the random type I problem instances with 35 sectors were found by solving the following integer programming problem:

$$Max \sum_{i<j} c_{ij} x_{ij} + \sum_{j<i} c_{ij} (1 - x_{ji})$$

subject to

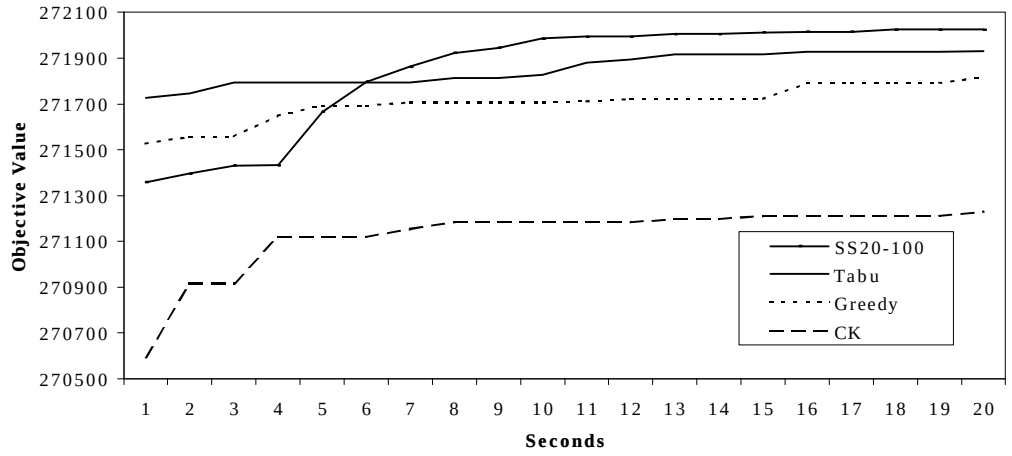
$$x_{ij} + x_{jk} - x_{ik} \leq 1 \quad \forall (i,j,k) : i < j < k$$

$$x_{ij} - x_{ik} + x_{jk} \geq 0 \quad \forall (i,j,k) : i < j < k$$

$$x_{ij} \in \{0,1\} \quad \forall (i,j) : i < j$$

In this formulation, x_{ij} = 1if sector *i* goes before sector *j* in the ordering, and 0 otherwise. The problems were solved with CPLEX 6.0 on a Pentium machine with 450 Mhz. The minimum time required was 101 seconds with a maximum of 13.3 hours and an average of 1.5 hours.

Figure 5. Performance graph for four procedures.



Finally, Figure 5 shows a performance graph that compares four procedures using 10 random type I instances of size 100. The procedures were run for 20 seconds and the best solution found was reported every second. The greedy procedure is performed from random starts. Note that SS20-100 uses the first 3 seconds to generate the set P and construct the *RefSet*. When the optimization process starts, the procedure quickly moves to the range of high quality solutions and maintains its lead during the rest of the solution time.

5. Conclusions

The objective of our study has been to expand and advance the knowledge associated with the implementation of scatter search procedures. Unlike its close cousin, tabu search, scatter search has not yet been extensively studied. In particular, we have undertaken to examine the critical issue of what form of diversification proves effective in this setting. We have also exploited a definition of quality that depends not only on an objective function value, or even on the properties of a given solution in isolation, but which considers the relative differences of the solutions maintained in the reference set. Finally, we have generated solution statistics that show the origins of the best solutions, both in terms of the types of combinations that produced them and in terms of the ranks of the ancestor solutions. Our resulting scatter search implementation is highly effective and rivals the best procedures in the literature.

References

Becker, O. (1967) “Das Helmstädtersche Reihenfolgeproblem — die Effizienz verschiedener Näherungsverfahren” in *Computer uses in the Social Sciences*, Berichteiner Working Conference, Wien, January 1967.

Chanas, S. and P. Kobylanski (1996) "A New Heuristic Algorithm Solving the Linear Ordering Problem," Computational Optimization and Applications, vol. 6, pp. 191-205.

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.

Feo, T. and M. G. C. Resende (1995) "Greedy Randomized Adaptive Search Procedures," *Journal of Global Optimization*, vol. 2, pp. 1-27.

Fisher, H. and G. L. Thompson (1963) "Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules," *Industrial Scheduling*, J. F. Muth and G. L. Thompson (eds.) Prentice-Hall, pp. 225-251.

Glover, F. (1965) "A Multiphase Dual Algorithm for the Zero-One Integer Programming Problem," *Operations Research*, vol. 13, no. 6, pp. 879-919.

Glover, F. (1968) "Surrogate Constraints," *Operations Research*, vol. 16, pp. 741-749.

Glover, F. (1998) "A Template for Scatter Search and Path Relinking," in *Artificial Evolution, Lecture Notes in Computer Science* 1363, J.-K. Hao, E. Lutton, E. Ronald , M. Schoenauer and D. Snyers (Eds.), Springer, pp. 13-54.

Glover, F. and M. Laguna (1997) *Tabu Search*, Kluwer Academic Publishers, Boston.

Grotschel, M., M. Junger and G. Reinelt (1984), "A Cutting Plane Algorithm for the Linear Ordering Problem," *Operations Research*, vol. 32, no. 6, pp. 1195-1220.

Knuth, D. E. (1993) *The Stanford GraphBase: A Platform for Combinatorial Computing*, Addison Wesley, New York.

Laguna, M. and F. Glover (1993) "Integrating Target Analysis and Tabu Search for Improved Scheduling Systems," *Expert Systems with Applications*, vol. 6, pp. 287-297.

Laguna, M., R. Martí and V. Campos (1998) "Intensification and Diversification with Elite Tabu Search Solutions for the Linear Ordering Problem," to appear in *Computers and Operations Research*.

LOLIB (1997) http://www.iwr.uni-heildelberg.de/iwr/comopt/soft/LOLIB/LOLIB.html.

Reinelt, G. (1985) *The Linear Ordering Problem: Algorithms and Applications*, Research and Exposition in Mathematics, Vol. 8, H. H. Hofmann and R. Wille (Eds.), Heldermann Verlag Berlin.