Scatter Search for the Profile Minimization Problem

Jesús Sánchez-Oro
Departamento de Ciencias de la Computación, Universidad Rey Juan Carlos, Madrid, Spain

Manuel Laguna
Leeds School of Business, University of Colorado at Boulder, Boulder, Colorado

Abraham Duarte
Departamento de Ciencias de la Computación, Universidad Rey Juan Carlos, Madrid, Spain

Rafael Martí
Departamento de Estadística e Investigación Operativa, Universidad de Valencia, Valencia, Spain

We study the problem of minimizing the profile of a graph and develop a solution method by following the tenets of scatter search. Our procedure exploits the network structure of the problem and includes strategies that produce a computationally efficient and agile search. Among several mechanisms, our search includes path relinking as the basis for combining solutions to generate new ones. The profile minimization problem (PMP) is NP-Hard and has relevant applications in numerical analysis techniques that rely on manipulating large sparse matrices. The problem was proposed in the early 1970s but the state-of-the-art does not include a method that could be considered powerful by today's computing standards. Extensive computational experiments show that we have accomplished our goal of pushing the envelope and establishing a new standard in the solution of the PMP. **© 2014 Wiley Periodicals, Inc. NETWORKS, Vol. 65(1), 10–21 2015**

Keywords: profile minimization; SumCut problem; metaheuristics; scatter search; OptQuest; LocalSolver; sparse matrices

1. INTRODUCTION

Given a graph $G(V, E)$—where V is a set of n vertices and E is a set of edges—an ordering (or permutation) φ of the vertices is a one-to-one mapping between the set $\{1, 2, \ldots, n\}$ and V. An ordering in a graph can be conceptualized as locating its vertices in a line, as shown in Figure 1. For a given ordering φ, the profile $\delta_{\varphi(i)}$ of vertex $\varphi(i)$—that is, the vertex in position i—is given by $\delta_{\varphi(i)} = i - f_{\varphi(i)}$, where $f_{\varphi(i)}$ is the smallest $j < i$ such that $\varphi(j)$ is adjacent to $\varphi(i)$ (if none exists then $f_{\varphi(i)} = i$). The profile $P(G, \varphi)$ of G with the ordering φ, is the sum of the profiles of all its vertices. The profile minimization problem (PMP) consists of finding an ordering φ^* of V such that the profile is minimized. Mathematically, the PMP seeks φ^*, over the set Φ of all possible permutations, such that

$$P(G, \varphi^*) = \min_{\varphi \in \Phi} \sum_{i=1}^{n} \delta_{\varphi(i)} = \min_{\varphi \in \Phi} \sum_{i=1}^{n} (i - f_{\varphi(i)})$$

Figure 1 shows an example of a network with six vertices ordered in a line, where the first one is labeled A, the second one B, and so on. In this figure, $f_1 = 1$ because there is no vertex $\varphi(j)$ for which $j < 1$. As a result, $\delta_1 = 0$. Similarly, $f_2 = 2$ and $\delta_2 = 0$. Conversely, $f_3 = 1$ and $\delta_3 = 3 - 1 = 2$, as the smallest j is 1, corresponding to vertex A. All additional calculations are shown in Figure 1, resulting in a profile value of $P(G, \varphi) = 11$.

It is possible to reduce the profile of the graph in Figure 1 by permuting its vertices. Figure 2 shows the same graph with the ordering $\varphi^* = (B, D, F, E, C, A)$, which results in an optimal profile value of $P(G, \varphi^*) = 8$.

Another interpretation of the PMP on a graph is based on the notion of labeling. Each vertex v in the graph is assigned a label $\pi(v)$ that is nothing else than the position that the vertex would occupy if the vertices were arranged in a line as in Figures 1 and 2. A solution is fully specified when all vertices have been labeled. By definition, the relationship between φ and π is such that if $\varphi(i) = v$ then $\pi(v) = i$. For a

Received May 2012; accepted September 2014
Correspondence to: M. Laguna; e-mail: laguna@colorado.edu
Contract grant sponsor: Ministerio de Educación y Ciencia of Spain; Contract grant number: TIN2012-35632-C02
DOI 10.1002/net.21571
Published online 23 December 2014 in Wiley Online Library (wileyonlinelibrary.com).

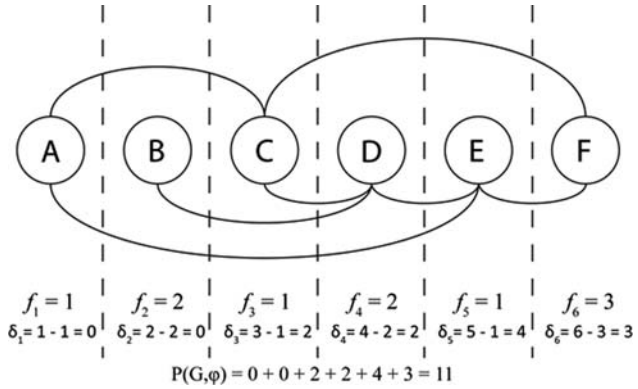

FIG. 1. Illustrative graph and profile calculation.

given π, the profile of vertex v is calculated as $\delta_v = \pi(v) - f_v$, where f_v is the smallest label of a vertex adjacent to v, as long as such label is smaller than $\pi(v)$. Otherwise, $f_v = \pi(v)$ and $\delta_v = 0$. The objective of the PMP is to find a labeling π^* for which the sum of all vertex profiles is minimized. To facilitate the description of our work, we will use both interpretations.

The PMP was originally proposed as an approach to reduce the space requirements for storing sparse matrices [31]. In this context, the PMP was shown to be equivalent to the SumCut problem [1]. One of the main applications of the PMP continues to be the reduction of the space requirements to store systems of equations. Additionally, the PMP enhances the performance of operations on systems of nonlinear equations, such as the Cholesky factorization [30]. Another interesting application of the PMP is described by Karp [15] in the context of the Human Genome Project. The main goals of this project are to identify all genes in human DNA (between 20 and 25 thousand, approximately) and determine the sequences of the approximately 3 billion chemical base pairs associated with human DNA.

In archeology [16], the PMP has been used in connection with the problem of organizing items such as fossils, tools, and jewels according to a specific order. This process is known as "seriation" and consists of placing several items from the same culture in a chronological order determined by a method of establishing dates that are relative to each other. This results in a problem for which the reordering of the rows and columns of a matrix is required, which is equivalent to the reordering of a linear graph. Other applications of the PMP can be found in information retrieval [2] and fingerprinting [15].

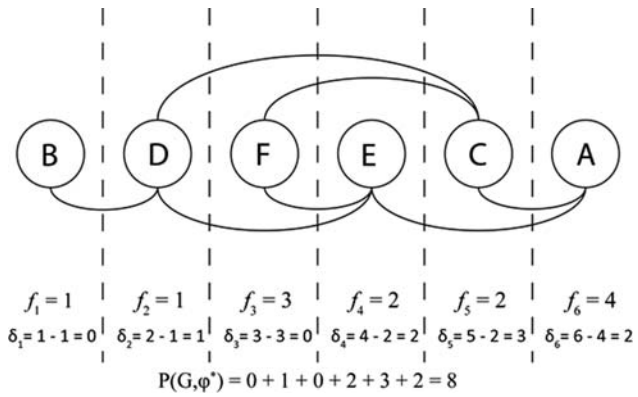

FIG. 2. Illustrative graph reordered.

Lin and Yuan [20] proved that the PMP for an arbitrary graph is equivalent to the interval graph completion problem, which was shown to be NP-complete by Garey and Johnson [10]. However, special classes of graphs can be solved optimally in polynomial time. For example, Lin and Yuan [20] proposed several polynomial-time algorithms to find the optimal solution of the PMP for paths, wheels, complete bipartite graphs and D4-trees (trees with diameter 4). Likewise, Guan and Williams [14] developed an algorithm to find the optimal solution of the PMP for triangulated graphs.

The SumCut problem [11, 25, 26] and the PMP are equivalent in the sense that a solution to one problem provides a solution to the other problem by reversing the corresponding permutation. Consequently, the optimum of one problem corresponds to the optimum of the other [1]. The PMP is also related to the bandwidth minimization problem (BMP), which consists of finding a permutation of the rows and the columns in a matrix, such that all the nonzero elements are confined to a band that is the closest to the main diagonal. The tabu search by Campos et al. [3] is the best BMP heuristic in the literature and we adapt it to the PMP to show that a specialized PMP search is justified to find solutions of the highest quality.

Algorithmically, the PMP has been tackled as the late sixties. To the best of our knowledge, the first heuristic in the literature consists of a constructive procedure proposed by Cuthill and McKee [4] known as the Reverse Cuthill–McKee algorithm (RCM). Their procedure is based on constructing a level structure of the vertices. Poole et al. [27] improved the RCM algorithm by changing the selection of the root node within a more general level structure. The method is known in the literature as GPS. Gibbs [12] developed GK, a procedure that uses a pseudodiameter to produce a new level structure. GK outperforms GPS in solution quality but it requires more computational time. Lewis [18] introduced new implementations that improve the performance of both GK and GPS. His experimental results confirm the superiority of GK over GPS in terms of solution quality.

The best heuristic for the PMP in the literature belongs to Lewis [19]. It uses the simulated annealing (SA) methodology in combination with existing constructive procedures. The SA search starts from a solution constructed with RCM or GK, whichever is better according to the objective function value. In typical SA fashion, neighborhood exploration is performed with moves that are randomly generated. Improving moves are always executed and nonimproving moves are only executed with a probability that depends on the current temperature. A so-called cooling schedule controls the systematic reductions of the temperature and the search ends when the temperature reaches a prespecified minimum level.

In contrast to all past efforts, we develop a specialized procedure that it is built within the scatter search (SS) framework.

Our main contribution consists of designing SS methods that are effective in finding high-quality solutions to the PMP. The mechanisms that we develop exploit the network structure of the problem and may be adapted to problems with similar characteristics. For instance, our diversification generation procedure creates diverse solutions by a strategy that is based on labeling the vertices in the graph. Likewise, the strategies embedded in the improvement method use label exchanges to explore the neighborhood of the current solution. A careful analysis of the neighborhood structure is performed to design an efficient process for evaluating exchanges. Finally, the combination method creates paths between solutions that result in a series of relabeling steps from an initiating to a target graph.

Our computational experiments and statistical tests show that we have been able to establish new benchmarks for the PMP. We compare not only against the best methods in the literature but also against a state-of-the-art procedure for a closely related problem (the BMP) and against two general-purpose black-box commercial optimizers with embedded metaheuristic search engines.

2. SCATTER SEARCH

Scatter search [17] is a metaheuristic whose framework includes five methods that are designed to build, maintain, and transform a population of solutions (see Fig. 3). Three of these methods, the diversification generation, the improvement, and the combination methods, are problem dependent and therefore their strategies take advantage of information that is context specific. Conversely, the Reference Set Update and the Subset Generation Methods have standard implementations that are context independent. The SS literature is fairly rich and includes multiple examples of successful applications, such as those documented in Gallego et al. [9], Martí et al. [21], Duarte et al. [7], and Pantrigo et al. [24].

The procedure starts with the application of the diversification generation method (which we describe in section 3 in the context of the PMP) and the Improvement method (described in section 4), obtaining as a result a population P of solutions from which the initial reference set (*RefSet*) of size b is constructed. The initial *RefSet* must balance solution quality and diversity and therefore the standard update in step 3 (Fig. 3) selects the best $b/2$ solutions from P and then the $b/2$ solutions in $P \setminus RefSet$ that are most diverse with respect to those solutions already in the reference set. We point out that selecting the most diverse solutions from a set is an NP-hard problem (see for instance Duarte and Martí [6]) and hence we perform this step heuristically. In particular, after selecting the $b/2$ best solutions (i.e., the ones with the best objective function value), the remaining solutions are added one at a time. The first one to be added is the solution in $P \setminus RefSet$ that is the most diverse with respect to the solutions currently in *RefSet*. Diversity is typically measured with a function that maximizes the minimum distance between the solution under consideration and the set of solutions where the solution will be added (in this case *RefSet*). As discussed above, a solution s to the PMP is fully characterized by its ordering φ_s and equivalently by the labeling π_s of the vertices in G. The distance between s and *RefSet* is given by

$$d(s, RefSet) = \min_{r \in RefSet} \left(\sum_{v=1}^{n} |\pi_s(v) - \pi_r(v)| \right)$$

The distance is the sum of the absolute differences of the labels assigned to each vertex in the candidate solution s and all the reference solutions r. As the labels represent positions, the calculation is equivalent to the so-called positional distance [5]. Once a solution s is selected from $P \setminus RefSet$, the solution is added to *RefSet* and the process is repeated $b/2$ times, choosing at each step the solution s^* with the maximum distance to the solutions currently in the reference set, that is:

$$s^* = \arg \max_{s \in P \setminus RefSet} d(s, RefSet)$$

In our implementation, step 4 in Figure 3 consists of generating all pairs of reference solutions that have not been combined before. Details about the combination method [based on the path relinking (PR) methodology] are presented in section 5.

The reference set update in step 7 is different from the updating performed in step 3. Assume that the reference solutions r are ordered according to their objective function values in such a way that the solution $r^{(1)}$ in the first position in the *RefSet* is the best and the solution $r^{(b)}$ in the last position is the worst. To maintain the diversity among the reference solutions, a new solution s is admitted if either of the following conditions is satisfied, where $P(s)$ is the profile of solution s:

1. $P(s) < P(r^{(1)})$
2. $P(s) < P(r^{(b)})$ and $d(s, RefSet) > dthresh$

The first condition establishes that a new solution becomes a reference solution if it is better than the best solution found so far. The second condition allows a new solution s into the reference set if it is better than the worst reference solution and its distance to the current reference set is larger than the distance threshold parameter *dthresh*. If solution s is admitted to the reference set, then s replaces the closest (according to d) reference solution r from all those for which $P(s) < P(r)$.

```
1. Diversification generation
2. Improvement
3. Reference set update
while (termination criteria not satisfied)
    4. Subset generation
    5. Combination
    6. Improvement
    7. Reference set update
```

FIG. 3. Scatter search framework.

The process terminates when no new solutions become part of the *RefSet*. That is, if at a given iteration, *RefSet* does not change after step 7, then the search ends. We point out that our step 7 update is significantly different from the one typically used in SS implementations, where a new solution s replaces $r^{(b)}$ as long as $P(s) < P(r^{(b)})$.

3. DIVERSIFICATION GENERATION METHOD

We develop four diversification generation methods (labeled C1–C4) to build a population P of solutions. These methods are based on the GRASP methodology [8, 28, 29]. To describe these methods we define U as the set of vertices that have not been labeled and $L = V \setminus U$ as the set of the vertices that have already been labeled. Initially, all vertices are in U (i.e., $U = V$). C1 starts by selecting the vertex $w \in U$ with the smallest degree (i.e., the vertex with the least number of adjacent vertices). In this step, ties are broken arbitrarily. The chosen vertex w is given the first label and therefore $\pi(w) = 1$ and $\varphi(1) = w$. Then $U = U \setminus \{w\}$ and $L = L \cup \{w\}$ and a candidate list CL consisting of the set of vertices adjacent to w is constructed:

$$\mathrm{CL} = \{u : (w, u) \in E\}$$

A greedy function value is calculated for each vertex v in CL as follows:

$$g_1(v) = |N_L(v)| - |N_U(v)|$$

where $N_L(v) = \{u \in L : (v, u) \in E\}$ and $N_U(v) = \{u \in U : (v, u) \in E\}$. The greedy function $g_1(v)$ measures the level of "urgency" of labeling vertex v next. The function considers that it is more urgent to label a vertex for which most of its adjacent vertices have already been labeled. A greedy procedure would choose, at each step, the vertex v with the largest $g_1(v)$ value. However, in the GRASP framework, constructions are semigreedy and the implementation includes a so-called restricted candidate list (RCL). The RCL includes top candidates that are equally preferred and hence equally likely to be chosen:

$$\mathrm{RCL} = \{v \in \mathrm{CL} : g(v) \leq g_1^{\min} + \alpha_1(g_1^{\max} - g_1^{\min})\}$$

where $g_1^{\min}$ ($g_1^{\max}$) is the minimum (maximum) value of $g_1(v)$ for all v in CL, and α_1 is a parameter that controls randomness/greediness of the corresponding procedure. In particular, if $\alpha_1 = 0$, the constructive method would be deterministic and completely greedy. Conversely, for $\alpha_1 = 1$ the procedure becomes totally random. A vertex v from RCL is chosen randomly, the next available label is assigned to it, and the U and L sets are updated. CL is also updated by adding the unlabeled vertices that are neighbors of the selected vertex v (i.e., $\mathrm{CL} = \mathrm{CL} \cup N_U(v)$). The construction ends when all vertices have been labeled.

The second construction procedure (C2) is similar to C1 but implements the semigreedy selection in a different way. Instead of calculating the greedy function value first and then randomly selecting from a restricted candidate list, C2 first takes from CL a random sample of vertices. Then, the vertex with the largest $g_1(v)$ value in the sample is selected. The size of the random sample is controlled by the parameter α_2, which represents a fraction of the size of the candidate list (i.e., the random sample includes $\alpha_2|\mathrm{CL}|$ vertices). As before, once the vertex is selected, the next available label is assigned to it and the U and L sets are updated.

The $g_1(v)$ greedy function does not take into consideration the values of the labels assigned to the vertices adjacent to v. For instance, if i is the next available label, $g_1(v)$ does not include in its calculation the labels that vertices in $N_L(v)$ have received (which may be any between 1 and $i-1$). Clearly, the "urgency" of vertex v is greater if its adjacent vertices have received labels that are much smaller than i. The following greedy function takes this into consideration:

$$g_2(v) = (|N_L(v)| - |N_U(v)|) \sum_{u \in N_L(v)} |\pi(v) - \pi(u)|$$

This greedy function is used to formulate two additional construction methods. C3 is the same as C1 but with g_2 as the greedy function. C4 is the same as C2 but with g_2 as the greedy function.

4. IMPROVEMENT METHOD

For our improvement method we tested both a search neighborhood defined by swap moves and one defined by insert moves. The representation of a solution as an ordering of the vertices is useful to conceptualize these search neighborhoods. A $swap(i, j)$ is the exchange of positions of vertices v, u that are currently in positions i and j, respectively. That is, $\varphi(i) = v$ and $\varphi(j) = u$. An $insert(i, j)$ is the movement of vertex $v = \varphi(i)$ to position j. After the move, v precedes u if $i > j$ and v follows u if $i < j$. As both of these moves produce a neighborhood of size $\mathcal{O}(n^2)$, a fast calculation of the move value is critical to search such a neighborhood efficiently and identify the best move to make.

For instance, to evaluate the move insert(i, j) for $j > i$, it is easy to see that only the profile of the vertices v for which $i \leq \pi(v) \leq n$ needs to be recalculated. Therefore, if the profile values for all vertices in the current solutions are stored, then the calculation of the move value may be done by updating only those relevant profile values and adding them to the values that the move does not affect. Additionally, the updates to the profile values may be accumulated. For instance if we first evaluate the insert (i, j), then $n-i+1$ values must be recalculated, i.e., those corresponding to the vertices with labels between i and n. If we then evaluate insert$(i, j+1)$ without storing any information from the previous evaluation, we would need to recalculate once again $n - i + 1$ values. However, observing the change of the profile values from one trial move to the other, it can be determined that the only changes occur in the vertices with labels $\pi(v) \geq j + 1$.

Figure 4 illustrates two insertion moves applied to solution $\varphi = \{A, B, C, D, E, F\}$, on the left side of the figure. The first

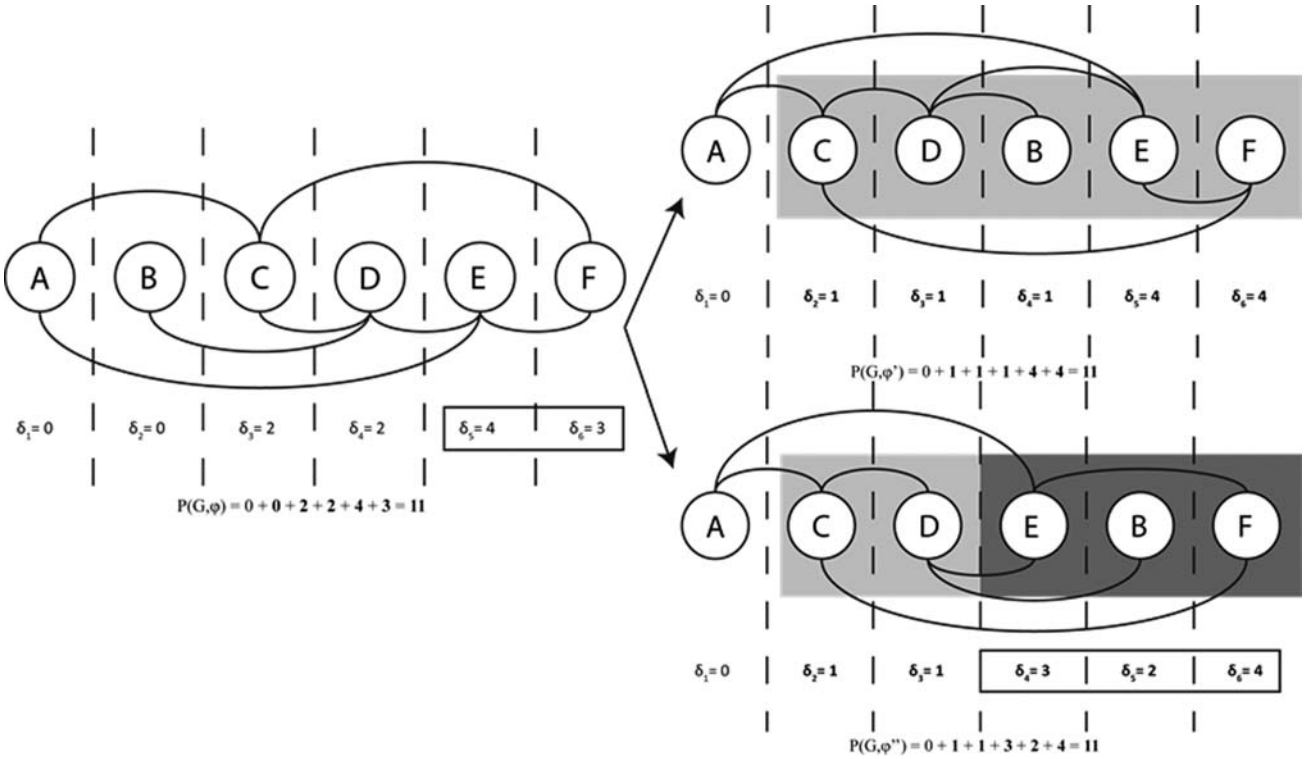

FIG. 4. Illustrative representation of a move.

move inserts vertex B in position 4, obtaining the solution $\varphi' = \{A, C, D, B, E, F\}$ at the top right of Figure 4, where vertices that changed their δ_i value are highlighted (i.e., vertices C, D, B, E, and F). The insertion of B in position 5 that yields a new solution φ'' is depicted at the bottom right of Figure 4. Considering that the improvement procedure evaluates insertions as a sequence of swap moves, evaluating the insertion of B in position 5 after evaluating the insertion of B in position 4 requires the updating of only vertices E, B, and F. Calculation savings are achieved because there is no need to update C and D again.

Therefore, an efficient way of searching the insert neighborhood is to evaluate the insert(i, j) as a sequence of swaps of the vertices in positions $(i, i+1)$, then $(i+1, i+2)$ and so forth until $(j-1, j)$. These values are stored because the sequence is the same for insert$(i, j+1)$ with the addition of the swap$(j, j+1)$. We have implemented this strategy and the corresponding one for the case when $j < i$.

By implementing the search as a sequence of swaps of vertices in adjacent positions, the only possible moves values are -1, 0, and +1. Consider the $swap(i, i+1)$ that exchanges the labels of vertex v from $\pi(v) = i$ to $i+1$ and vertex u from $\pi(u) = i+1$ to i. Then, the move value is given by

$$\text{value}(i, i+1) = \begin{cases} -1 & \text{if } f_v > i \text{ and } f_u < i+1 \\ 0 & \text{if } (f_v > i \text{ and } f_u \geq i+1) \\ & \quad \text{or } (f_v \leq i \text{ and } f_u < i+1) \\ +1 & \text{if } f_v \leq i \text{ and } f_u \geq i+1 \end{cases}$$

This characteristic renders a first improving strategy impractical. A first improving refers to the strategy of stopping the neighborhood search after finding the first move that improves the current solution. As our neighborhood search is structured in such a way that complex moves [i.e., insert(i, j) for $j > i+1$] are achieved by a sequence of simple moves [i.e., swap$(i, i+1)$], stopping after a finding any improvement results in a process where improvements of more than one unit are not possible in a single move. Therefore, our implementation uses the best improving strategy in which the entire neighborhood is searched and the move with the best value is selected.

5. COMBINATION METHODS

We have developed one combination method (CM1) and a variant (CM2). The combination method follows the strategy known as PR [13]. The main idea behind PR is to construct a path between two elite solutions where the guide is not limited to the value of the objective function of the problem. PR was suggested in connection with tabu search because choice mechanisms in neighborhood-based searches often use the objective function as the only oracle to measure the quality of a move (just as we do in the improvement method described above). PR attempts to create search paths where the objective function is only one of the elements used to determine the direction. PR also exploits the principle that the neighborhood of an elite solution might contain other high-quality solutions that could be found if the elite solution is approached from a direction that is different from the one that was used to

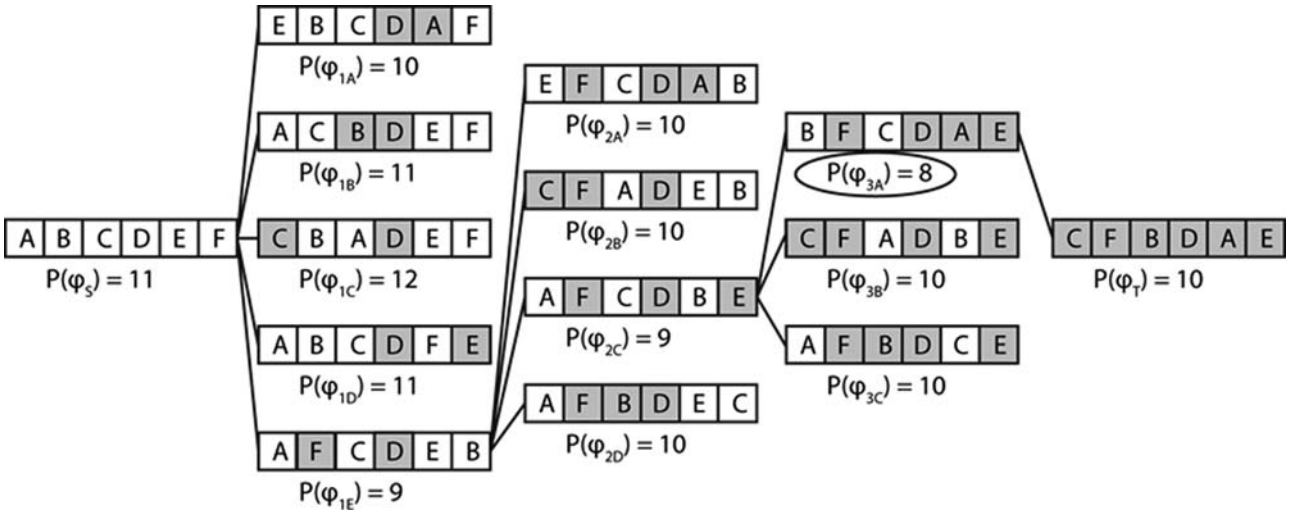

FIG. 5. Illustration of the combination method CM1.

find the elite solution in the first place. PR is implemented by choosing one or more elite solutions as the initiating solution and one or more as the guiding solutions. Strategies are then built around the notion of applying transformations to the initiating solution with the goal of moving it toward the guiding solution(s). As all solutions that have been found during a search are connected by the path that the search followed to find them, building a new path between elite solutions may be conceptualized as a relinking exercise, and hence the name of the strategy.

Let s be the initiating (or starting) solution and t the guiding (or target) solution. CM1 starts by identifying the set D of vertices that have different labels in s and t, that is, $D = \{v : \pi_s(v) \neq \pi_t(v)\}$. A set of solutions is generated by swapping the labels of vertex v and vertex $\varphi_s(\pi_t(v))$ in the initiating solution, for all $v \in D$. Note that after these swaps, $\pi_s(v)$ equals $\pi_t(v)$ and the initiating solution moves closer to the guiding solution. This step generates $|D|$ trial solutions from which we select the best according to the objective function value. The chosen solution becomes the new initiating solution and the process continues until $D = \emptyset$, that is, until the labeling in s is the same as in t. The procedure, referred to as Greedy Path Relinking in [28], returns the best trial solution found.

Figure 5 illustrates how CM1 operates on two solutions, where s is the initiating solution and t is the guiding solution. The relinking requires 4 steps and generates 12 intermediate solutions that are labeled with a digit and a letter, where the digit is the step number and the letter a solution identifier. Note that at each step, the best solution (according to the objective function value) becomes the initiating solution for the next step. Once the guiding solution is reached, the best intermediate solution (in this case solution 3A with a profile value of 8) is returned.

We created a variant of CM1 that we refer to as CM2 and that consists of replacing the selection criterion of the trial solution generated during the relinking process. In particular, instead of selecting the best solution with respect to the objective function value from the set D of trial solutions, the next initiating solution is selected randomly. In this way, CM2 favors diversification over intensification. As before, CM2 returns the best solution encountered during the PR process.

6. COMPUTATIONAL EXPERIMENTS

We now describe the computational experiments performed to test the SS approach that we developed for the PMP and then we discuss the resulting outcomes. The SS procedure was implemented in Java SE 6 and was compared against the best methods reported in the literature so far, namely, RCM [4], and SA [19]. The RCM and SA results were obtained running the original C implementations shared by the authors. We also compare against an adaptation of the best heuristic for the BMP as well as two general-purpose optimizers. All tests were performed on an Intel Core i7 2600 machine running at 3.4 GHz and with 2 GB of RAM. The test set consists of 262 instances divided into three subsets. All instances are available at www.optsicom.es/pmp/.

- HB—This set is derived from 73 instances in the Harwell-Boeing Sparse Matrix Collection [22]. The data matrices in this set correspond to problems in linear systems, least squares and eigenvalue calculations in a wide variety of scientific and engineering disciplines. We selected the 73 problems with $n \leq 1000$ and therefore the number of vertices ranges from 24 to 960 and the number of edges ranges from 34 to 3721.
- K-Graphs—This set contains 98 bipartite graphs with number of vertices ranging from 4 to 142 and number of edges ranging from 3 to 5016. A complete bipartite graph is such that the set of vertices V can be divided into two subsets V_1 and V_2 in such a way there exists an edge between every pair of vertices, one belonging to V_1 and the other belonging to V_2. At the same time, there is no edge for which its endpoint vertices are in the same subset. Optimal objective function values are known by construction [20] and are given by $n_1 n_2 + \frac{1}{2} n_1 (n_1 - 1)$ for $n_1 \leq n_2$, where $n_1 = |V_1|$ and $n_2 = |V_2|$.

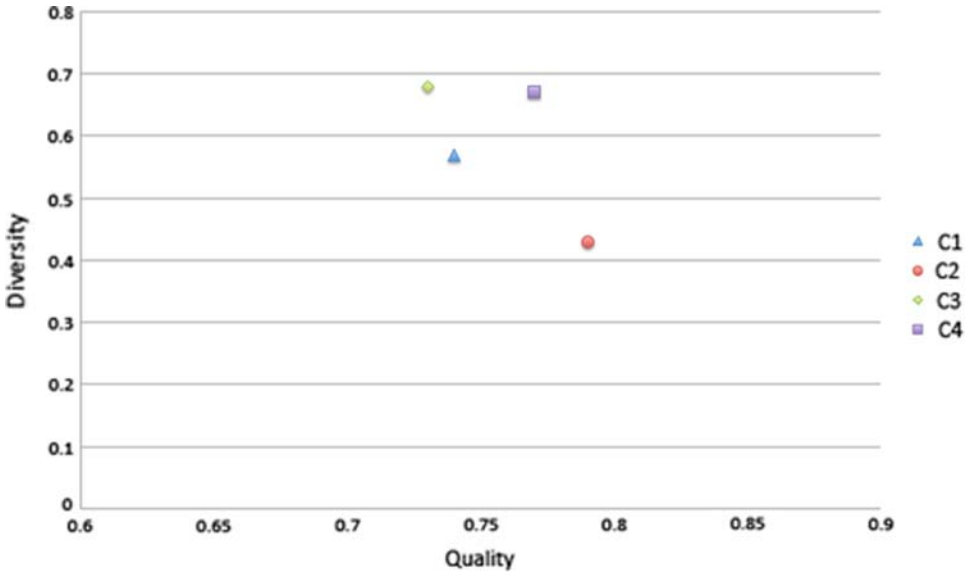

FIG. 6. Comparison of construction methods. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

- D4-Trees—This set consists of 91 instances that are based on trees with a diameter of 4 and a number of vertices ranging from 10 to 100 and a number of edges ranging from 9 to 99. These trees are built by choosing a root vertex v_0 and adjacent vertices v_1 to v_k, for $k \geq 2$. The vertices adjacent to v_0 form a star with v_0 in the center. If $\gamma(v_i)$ is the degree of vertex v_i then the v_i-branch has $\gamma(v_i) - 1$ leaves, because the other edge is the one connecting v_i to the root vertex v_0. The trees in the D4 set are built in such a way that $\gamma(v_1) \geq \gamma(v_2) \geq \cdots \geq \gamma(v_k) \geq 2$ and with a diameter of 4. This means that the tree consists of the root vertex, of all the adjacent vertices in a star configuration and of all the vertices corresponding to the leaves of the tree. The optimal objective function values are known by construction and are given by $|E|+\sum_{i=3}^{k}(\gamma(v_i)-1)$.

The experiments are divided into two main blocks. The first block has the goal of studying the behavior of the components of the solution procedure as well as determining the best values for the search parameters. The second block of experiments has the goal of comparing our procedure with the best in the literature and two commercial optimizers. To be able to test the effectiveness of our strategies in the entire PMP class, the first set of experiments is performed on a subset of all problem instances. In this way, we can test how well our choices generalize to the entire set of problems. Specifically, the tuning experiments are performed on the 30 HB instances that have less than 250 vertices. We refer to these instances as the training set and to all other instances as the testing set.

A common practice in SS implementations is to arrive to the best design by choosing the components of the solution procedure sequentially. As we are using standard implementations for the reference set update and the subset generation methods, our design process focuses on choosing a diversification generator, an improvement method and a combination method. As described in the previous sections, we have developed four diversification generation methods (C1–C4), two improvement methods (one based on swaps and one based on inserts), and two combination methods (CM1 and CM2). This results in a full-factorial design with 16 combinations. We start by contrasting the sequential process that selects components one at a time and adds them to the solution procedure with the full-factorial approach that identifies the best combination with a single experiment.

For the sequential design process, we begin by comparing the four construction methods described in section 3. As these methods are used for diversification purposes, the comparison metrics must include a measure of diversity as well as a measure of solution quality. Each method (C1–C4) is executed to generate 100 solutions. The value of α_1 for C1 and C3, and α_2 for C2 and C4 are chosen at random in each construction. For validation purposes, we also generate 100 solutions at random and refer to this set of solutions as C0. The objective function value of each solution is used to calculate an average quality of each set, which is then normalized to a value between 0 and 1. The average diversity of a set of solutions is calculated using the distance measure described in section 2. That is, for a particular set of solution, the distance between a solution and the rest in the set (i.e., the other 99 solutions) is computed and then the average of the 100 values obtained is used to represent the diversity of the set. We also normalize this diversity measure to obtain a value between 0 and 1. Figure 6 compares the diversity and quality of the construction methods.

Not surprisingly, C0 obtains the largest diversity of all the sets, as well as the lowest quality, and hence it is not depicted in the figure. However, this set of solutions is of minimum quality when compared to the constructions based on GRASP. The C2 set has the highest quality but the lowest diversity. The sets generated by C3 and C4 are very similar, with C3 slightly more diverse and C4 with slightly higher quality. To

TABLE 1. Construction procedures coupled with improvement methods

Variant	Obj	Dev (%)	#Best	Time
C2+Swap	1235.94	4.61	7	68.76
C3+Swap	1209.72	2.05	10	90.84
C2+Insert	1209.34	1.30	17	2.56
C3+Insert	1194.78	0.67	18	3.90

test the effect of adding the improvement method to these construction procedures, we select C2 and C3 to perform additional experiments. Ignoring the purely random generator C0, the set generated by C2 and C3 are at the extremes of the frontier in which C1 is dominated and C4 is the most balanced nondominated point.

The second experiment couples the construction methods C2 and C3 with the improvement methods based on swap and insert neighborhoods, resulting in 4 different variants. Once again, the α values associated with C2 and C3 are chosen randomly in each construction. Table 1 shows results obtained when applying these variants to the training set. For each variant, the table reports the average objective function value (Obj), the percent deviation of this value from the average obtained with the best known solutions (Dev), the number of best solutions found out of 30 (#Best) and the CPU time in seconds.

The improvement method based on inserts seems to be more effective than the one based on swaps according to the results in Table 1. The table also shows that local search based on swap moves is computationally more expensive than the one based on insert moves. The best combination of construction and improvement is given by C3+Insert, CPU time notwithstanding. Therefore, in this sequential process, we have determined, so far, that our final SS should have C3 as the diversification generator and Insert as the improvement method.

Finally, we must choose a combination method that works well with the C3+Insert improvement method. For this experiment, we also need to add the subset generation method and the reference set improvement method. The subset generation method is a standard implementation and has no parameters. The behavior of the reference set update method depends on the values of b and the *dthresh* parameters. Although we set b to the standard value of 10 used in the SS literature, in this experiment, we try 4 values for *dthresh* (0.01, 0.05, 0.10, and 0.30), which is given as a fraction of the maximum distance MD between the labeling of two solutions. Considering the distance function formulated in section 2, *MD* is computed as follows:

$$\mathrm{MD} = \sum_{i=1}^{n} |i - (n - i + 1)|$$

As indicated at the end of section 2, the search terminates when no solution generated by the application of the combination and improvement methods is admitted to the reference set. Table 2 summarizes the results of this experiment, where

TABLE 2. Performance of combination methods CM1 and CM2

Variant	*dthresh*	Obj	Dev (%)	#Best	Time
	0.01* MD	1187.97	0.10	25	6.54
C3+Insert+CM1	0.05* MD	1187.38	0.08	25	6.37
	0.10* MD	1187.63	0.09	24	6.71
	0.30* MD	1188.16	0.12	24	6.73
	0.01* MD	1191.69	0.43	21	6.08
C3+Insert+CM2	0.05* MD	1191.56	0.43	21	5.74
	0.10* MD	1191.78	0.45	21	5.90
	0.30* MD	1191.56	0.43	20	5.43

the column labels have the same meaning as in Table 1 with the addition of the *dthresh* column containing the value of the parameter that was used to produce the results.

According to the results shown in Table 2, the best SS configuration should include CM1 as the combination method. Within that, the best value for *dthresh* seems to be 0.05, which results in the smallest deviation from the best known solutions. The results indicate that while there is a significant difference in performance between using CM1 and using CM2, the procedure is robust with respect to the value of *dthresh*. In particular, there is no significant difference between values in the range between 0.01 and 0.10 for CM1.

The final experiment of this block consists of choosing the best SS configuration by running a single full-factorial experiment. To perform this experiment, we set $b = 10$, $dthresh = 0.05$ and α is chosen randomly in each construction. Table 3 summarizes the results obtained with the 16 variants on the training set.

The results of the full-factorial design that Table 3 summarizes indicate that there are two configurations that dominate all others when considering percent deviation and number of best solutions: C1+Insert+CM1 and C3+Insert+CM1. The C3+Insert+CM1 configuration was the one identified as the best by the sequential design process. It achieves the lowest

TABLE 3. Full-factorial experiment

Variant	Obj	Dev (%)	#Best	Time
C1+Insert+CM1	1189.44	0.80	20	6.59
C1+Insert+CM2	1195.38	0.96	18	5.30
C1+Swap+CM1	1195.94	1.35	13	92.68
C1+Swap+CM2	1200.91	1.74	12	91.24
C2+Insert+CM1	1218.31	3.30	11	3.79
C2+Insert+CM2	1219.41	3.51	12	3.48
C2+Swap+CM1	1228.91	4.42	9	76.20
C2+Swap+CM2	1232.91	4.74	7	75.36
C3+Insert+CM1	1188.06	0.75	17	6.78
C3+Insert+CM2	1191.53	1.04	14	6.58
C3+Swap+CM1	1201.91	1.79	10	101.27
C3+Swap+CM2	1202.78	1.95	12	96.13
C4+Insert+CM1	1195.28	1.27	14	6.91
C4+Insert+CM2	1200.97	1.50	15	6.14
C4+Swap+CM1	1205.44	2.25	10	94.01
C4+Swap+CM2	1205.66	2.44	10	93.00

TABLE 4. Experiments with 98 K-Graphs

Procedure	Obj	Dev	#Opt	Time
RCM	1165.64	0.00	98	1.02
SA	1167.06	0.02	97	5.87
SS	1165.64	0.00	98	0.66

deviation of 0.75% and the third largest number of best solutions of 17. The C1+Insert+CM1 configuration achieves the largest number of best solutions of 20 and an average deviation of 0.8% that is the second lowest. The configuration uses a diversification generator that is different from the one that we selected during the sequential design process. In fact, in our analysis, C1 was dominated by the other three construction procedures in terms of both quality and diversity (see Fig. 6). Hence, the full-factorial design is able to identify a configuration that performs well when all elements are put together at once but that does not seem attractive when the merit of each element is assessed separately. Nonetheless, the fact that the C3+Insert+CM1 configuration is in the nondominated set seems to indicate that in situations where running a full-factorial design is not practical, a sequential process is an approach from which it is reasonable to expect an effective combination of SS components. This analysis concludes the first block of experiments.

In the second block of experiments, we start by comparing the performance of the SS procedure configured as C1+Insert+CM1 to the best methods in the literature. In particular, the comparison includes RCM [4] and SA [19]. Tables 4 and 5 show the results associated with the K-Graph and D4-Tree data sets, respectively. The optimal solutions to these problems are known by construction and therefore the deviations are calculated using the optimal objective function values. Also, the tables report the number of optimal solutions found (#Opt) instead of the number of best solutions.

Clearly, the K-Graphs do not represent a challenge to any of the procedures in our test. Only SA fails to find the optimal solutions to one of the instances, even when using considerably more time than its counterparts. The situation changes when the methods are applied to the D4-Tree data set. The results are shown in Table 5 where the difficulty of these problems becomes evident. The best performance is achieved by SS, which is able to find the optimal solution to 97.8% (i.e., 89 out of 91) of the problems. The solution time is negligible for all procedures.

We perform nonparametric tests to provide additional support to our conclusions about the performance of the SS implementation. First, we apply the Friedman test for multiple correlated samples to the best solutions obtained by each of the three methods in Table 5. This test computes, for each instance, the rank value of each method according to solution quality (where rank 1 is assigned to the best method and rank 3 to the worst). Then, it calculates the average rank values for each method across all instances. If the averages differ greatly, the associated p-value or level of significance is small. The resulting p-value obtained in this experiment (smaller than 0.001) clearly indicates that there are statistically significant differences among the three methods. The rank values produced by this test are 1.18 (SS), 1.85 (SA), and 2.97 (RCM).

TABLE 5. Experiments with 91 D4-Trees

Procedure	Obj	Dev(%)	#Opt	Time
RCM	282.75	173.57	2	1.01
SA	126.08	30.39	31	0.68
SS	86.12	0.01	89	0.28

Next, we used the Wilcoxon test and Sign test to make a pairwise comparison of SS and SA, which consistently provide the best solutions reported in our experiments. The results of the Wilcoxon test (with a p-value smaller than 0.001) determined that the solutions obtained by the two methods indeed represent two different populations. The Sign test (with a p-value smaller than 0.001) indicated that the objective function values of the solutions obtained with SS tend to be better (i.e., smaller) than those obtained with SA.

In the final experiment of this block, we add three procedures to our comparison set: (1) an adaptation of the tabu search developed by Campos et al. [3] for the solution of the BMP (TS-BMP), (2) OptQuest, and (3) LocalSolver. The BMP is related to the PMP because they both have the goal of finding a graph labeling such that the corresponding sparse incidence matrix is transformed into one for which the nonzero elements are close to the main diagonal. The PMP achieves this goal with an additive objective function that penalizes a distance measure from the main diagonal while the BMP uses a criterion that minimizes the maximum deviation. Specifically, in the BMP, the objective is to find an ordering of the rows and columns of a matrix M in such a way that the nonzero elements are in a band that is as close as possible to the main diagonal. In terms of the graph representation, the goal is to find a labeling π of the vertices of the corresponding $G(V, E)$ graph such that the bandwidth $B(G, \pi)$ is minimized, where:

$$B(G, \pi) = max(B(v, \pi) : v \in V) \quad \text{and}$$
$$B(v, \pi) = max(|\pi(v) - \pi(u)| : (v, u) \in E).$$

That is, $B(v, \pi)$ is the bandwidth of vertex v for labeling π and it is calculated as the maximum absolute difference between the label of v and the labels of its adjacent vertices. The bandwidth of the graph is the maximum vertex bandwidth. Given these similarities, it is reasonable to believe that a solution procedure designed for the BMP might find high-quality solutions to the PMP. To test this, we applied TS-BMP without any modifications to the HB instances. TS-BMP operates with the objective of minimizing $B(G, \pi)$ and on termination the procedure returns the solution π^* with the best value according to this objective function. We then calculate $P(G, \varphi^*)$ by applying the transformation $\varphi^*(\pi^*(v)) = v$ for all vertices in the graph.

TABLE 6. Experiments with 26 HB instances with $n \leq 200$

Procedure	Obj	Dev (%)	#Best	Rank	Time
OptQuest	1466.54	46.7	2	4.3	36.9
LocalSolver	1030.04	2.4	23	1.3	60.0
TS-BMP	1596.83	42.7	1	5.1	30.8
RCM	1109.58	20.4	2	3.9	97.3
SA	1110.92	14.5	4	3.3	4.0
SS	1007.58	1.1	15	1.5	2.8

TABLE 7. Experiments with 27 HB instances with $200 < n \leq 500$

Procedure	Obj	Dev (%)	#Best	Rank	Time
OptQuest	8923.07	61.1	0	4.1	211.9
LocalSolver	7492.30	25.5	8	2.9	220.0
TS-BMP	10204.82	73.4	0	5.2	212.1
RCM	7491.85	34.0	4	3.2	335.6
SA	7542.48	33.1	1	3.6	39.5
SS	5701.48	2.2	17	1.4	78.5

OptQuest (optquest.com) is a general-purpose black-box optimizer built on a SS platform. The system includes several variable types, including permutations. We used the latest OptQuest version in C# and followed three simple steps: (1) read data, (2) define optimization model by creating n permutation variables, and (3) create `Evaluate()` to evaluate the objective function. Each OptQuest iteration consists of the evaluation of a trial solution that is generated by the search mechanisms implemented within the system.

LocalSolver (localsolver.com) is also a black-box optimizer for combinatorial problems. LocalSolver uses a hybrid approach to optimization that ranges from neighborhood search and metaheuristics to constraint propagation and mathematical programming. The main elements of the search are the so-called autonomous moves (complex moves, such as ejection chains, that have the goal of preserving feasibility) and incremental evaluation (a concept introduced by Michel and van Hentenryck [23]). LocalSolver admits only binary variables and therefore a transformation must be used to apply it to an ordering problem such as the PMP. In particular, we created a LocalSolver model with binary variables x, such that `x[i][j]=1` indicates that vertex `i` has label `j`. Appendix A contains the LocalSolver Program (LSP) that we used to perform the experiment.

The results associated with this experiment are in Tables 6–8. As the optimal solutions to the HB instances are not known, the deviations are calculated against the best-known solutions and #Best refers to the number of best-known solutions found by each procedure. These tables also include the average rank for each procedure. The tables divide the instances into three different groups according to the number of vertices in the graph: 26 instances with less than or equal to 200 vertices (Table 6), 27 instances with more than 200 and no more than 500 vertices (Table 7), and 20 instances with more than 500 vertices (Table 8). As TS-BMP is designed to tackle connected graphs and 11 HB instances correspond to graphs that are not connected, our experiments with this code are limited to 62 instances (23, 22, and 17 instances for Tables 6–7, and 8, respectively).

The procedures from the literature (TS-BMP, RCM, and SA) were executed with the parameters values suggested by their authors. As all three methods are heuristics, we impose a maximum computing time of n seconds, where n, as before, represents the number of vertices. To stop the execution of OptQuest and LocalSolver, we also use a time limit that depends on the size of the problem. However, we allow them to run considerably longer on the larger instances because these two procedures are not specialized to the PMP. The stopping times (in seconds) are 60 ($n \leq 200$), 120 ($200 < n \leq 300$), 300 ($300 < n \leq 500$), 600 ($500 < n \leq 600$), 900 ($600 < n \leq 700$), 1200 ($700 < n \leq 800$), 1800 ($800 < n \leq 900$), 3600 ($n > 900$). We configure OptQuest to report the time when the incumbent solution is last updated during the run and those are the average times that we report in Tables 6–8. The time when the procedure stops is the same as the time reported for the LocalSolver.

The results in Table 6 indicate that LocalSolver is a very good alternative for problems with up to 200 vertices. This procedure finds the largest number of best solutions and therefore has the best rank. Its average deviation is slightly larger than SS due to a 42.58% deviation in problem CAN 198 (see Appendix B for the complete list of problems in the HB dataset). In contrast, the largest deviation for SS in this subset of problems is 16.06% (in the DWT 162 instance).

The merit of SS becomes evident as the problem size increases. The results in Tables 7 and 8 support the position that SS is the best procedure for the PMP when dealing with medium to large instances.

We applied statistical tests to the results in Table 6–8. The Friedman test detects significant differences in all cases (p-value smaller than 0.001). We used the Wilcoxon test and Sign test to make a pairwise comparison between the best two procedures in each set of instances. In particular, both tests are inconclusive with regard of LocalSolver and SS for the instances in Table 6. Conversely, considering SS and its closest competitor, LocalSolver (Table 7) and SA (Table 8), both tests indicate that the solutions obtained with SS are consistently and significantly (p-value smaller than 0.001) better, confirming the superiority of the proposed method. Table 9 in Appendix B shows the details of the SS output for the HB instances.

TABLE 8. Experiments with 20 HB instances with $n > 500$

Procedure	Obj	Dev (%)	#Best	Rank	Time
OptQuest	27156.00	105.4	1	4.7	1518.4
LocalSolver	26078.30	90.0	1	4.4	1575.0
TS-BMP	25226.29	61.4	0	4.5	714.7
RCM	19256.05	28.5	1	2.7	744.1
SA	19246.65	29.2	0	3.2	1367.3
SS	15629.05	1.0	17	1.2	674.8

TABLE 9. Best individual values on HB instances obtained with SS

Instance	Size	Best	SS	Time	Instance	Size	Best	SS	Time
CAN 24	24	95	**95**	0.1	CAN 292	292	4673	4718	41.7
BCSPWR01	39	82	**82**	0.1	DWT 307	307	6676	**6676**	46.4
BCSSTK01	48	460	466	0.4	DWT 310	310	2630	**2630**	25.0
BCSPWR02	49	113	**113**	0.3	DWT 346	346	6051	**6051**	54.1
DWT 59	59	214	223	0.4	DWT 361	361	4635	**4635**	64.8
CAN 61	61	338	**338**	0.4	PLAT362	362	9150	10620	105.8
CAN 62	62	172	**172**	0.5	LSHP 406	406	5964	**5964**	83.2
BCSSTK02	66	2145	**2145**	0.4	DWT 419	419	6679	**6679**	107.3
DWT 66	66	127	**127**	0.2	BCSSTK06	420	13239	13437	123.9
DWT 72	72	147	151	0.7	BCSSTK07	420	13224	13437	123.9
CAN 73	73	520	**520**	0.9	BCSPWR05	443	3076	3354	90.3
ASH85	85	490	**490**	1.4	CAN 445	445	15494	**15494**	150.4
DWT 87	87	428	434	1.2	NOS5	468	20446	**20446**	209.0
CAN 96	96	1078	1080	2.1	BCSSTK20	485	3006	**3006**	195.8
NOS4	100	651	**651**	1.1	DWT 492	492	3361	**3361**	151.0
BCSSTK03	112	272	**272**	0.3	494 BUS	494	3499	**3499**	237.5
BCSPWR03	118	434	**434**	3.2	DWT 503	503	13152	**13152**	192.3
BCSSTK04	132	3154	3159	4.9	DWT 512	512	3975	**3975**	144.6
BCSSTK22	138	628	641	2.2	LSHP 577	577	10035	10045	222.1
CAN 144	144	969	**969**	2.3	DWT 592	592	9498	**9498**	220.5
BCSSTK05	153	2191	2192	4.3	DWT 607	607	13278	**13278**	419.3
CAN 161	161	2482	**2482**	6.6	CAN 634	634	28493	**28493**	499.9
DWT 162	162	1108	1286	5.8	662 BUS	662	8962	**8962**	318.5
CAN 187	187	2184	2195	9.7	NOS6	675	9095	**9095**	184.2
DWT 193	193	4355	4388	15.3	685 BUS	685	8528	**8528**	799.8
DWT 198	198	1092	**1092**	7.6	CAN 715	715	24414	**24414**	984.3
DWT 209	209	2494	2621	25.1	NOS7	729	34226	34675	338.7
DWT 221	221	1646	**1646**	20.1	DWT 758	758	6392	**6392**	371.3
CAN 229	229	3928	4141	27.9	LSHP 778	778	15719	**15719**	586.2
DWT 234	234	782	803	10.6	BCSSTK19	817	7638	**7638**	855.7
NOS1	237	467	**467**	2.1	DWT 869	869	13107	**13107**	1526.5
DWT 245	245	2053	**2053**	29.7	DWT 878	878	17259	**17259**	1104.6
CAN 256	256	5049	**5049**	40.8	GR 30 30	900	24311	**24311**	1190.4
LSHP 265	265	3162	**3162**	25.7	DWT 918	918	16502	**16502**	1277.3
CAN 268	268	5215	**5215**	25.0	NOS2	957	1907	**1907**	36.0
BCSPWR04	274	1808	1992	40.6	NOS3	960	38676	45631	2222.8
ASH292	292	2717	2784	61.8					

7. CONCLUSIONS

Our goal was to develop a state-of-the-art solution method for the PMP. We accomplished this goal with an implementation of a SS procedure that computational experiments show to be superior to the solution methods reported in the literature. We developed a number of mechanisms that we believe can be helpful in similar problem settings. For instance, the ideas behind our efficient move evaluation in the improvement method could be adapted to other problems with similar characteristics (i.e., labeling vertices in a graph). Also, in the process of choosing the best SS design, we discovered that a sequential approach is capable of producing a highly competitive design and thus allowing future SS implementations to avoid a full-factorial design when such a design is computationally intractable.

APPENDIX A

This is the `model()` function used within the LSP applied to the HB instances that can be found in www.optsicom.es/pmp/. Note that `vcount[i]` contains the number of vertices adjacent to vertex `i` and `vlist[i][j]` is the `j`th vertex adjacent to vertex `i`. Also, `p` is the profile of each vertex that is being added to the value of `obj`.

```
function model()
{
  // x[i][j] equal to 1 if vertex i is assigned
     label j
  x[1..nodes][1..nodes] <- bool();

  // one label per vertex i for [i in 1..nodes]
     constraint sum[j in 1..nodes](x[i][j]) == 1;
  // one vertex per label j for [j in 1..nodes]
     constraint sum[i in 1..nodes](x[i][j]) == 1;

  //label[i] is the label of vertex i
  label[i in 1..nodes] <- sum[j in 1..nodes]
     (j*x[i][j]);
  obj <- 0; for [i in 1..nodes]
  {
    p <- 0;
    for[j in 1..vcount[i]] p <- max(label[i]
      - label[vlist[i][j]], p);
    obj <- obj + p;
  }
  minimize obj;
}
```

APPENDIX B

Table 9 shows the SS results for the 73 Harwell-Boeing instances (ordered by size). The table compares the SS objective function value and the best known value (Best) and it also shows the SS computing time (in seconds). Best solutions found by the SS procedure are shown in bold.

Acknowledgment

The authors would like to thank Professor Robert R. Lewis for sharing the source code of the RCM and SA methods.

REFERENCES

[1] A. Agrawal, P. Klein, and R. Ravi, Ordering problems approximated: Single-processor scheduling and interval graph completion, Automata Languages Programming 510 (1991), 751–762.

[2] R.A. Botafogo, Cluster analysis for hypertext systems, Proceedings of the 16th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval, 1993, pp. 116–125.

[3] V. Campos, E. Piñana, and R. Martí, Adaptive memory programming for matrix bandwidth minimization, Ann Oper Res 183 (2011), 7–23.

[4] E. Cuthill and J. McKee, Reducing the bandwidth of sparse symmetric matrices, Proc ACM 24th National Conf, 1969, pp. 157–172.

[5] A. Das and M. Roberts, Metric distances of permutations, 2004, Available at: www.cra.org/Activities/craw_archive/dmp/awards/2004/Das/paper.ps.

[6] A. Duarte and R. Martí, Tabu search and grasp for the maximum diversity problem, Eur J Oper Res 178 (2007), 71–84.

[7] A. Duarte, F. Glover, R. Martí, and F. Gortázar, Hybrid scatter tabu search for unconstrained global optimization, Ann Oper Res 183 (2011), 95–123.

[8] T.A. Feo and M.G.C. Resende, A probabilistic heuristic for a computationally difficult set covering problem, Oper Res Lett 8 (1989), 67–71.

[9] M. Gallego, A. Duarte, M. Laguna, and R. Martí, Hybrid heuristics for the maximum diversity problem, Comput Optim Appl 44 (2009), 411–426.

[10] M.R. Garey and D.S. Johnson, Computers and intractability: A guide to the theory of NP-completeness, W. H. Freeman and Co, 1979.

[11] A. Gibbons, M. Paterson, J. Torán, and J. Diaz, The minsumcut problem, Algorithms Data Struct 519 (1991), 65–79.

[12] N.E. Gibbs, A hybrid profile reduction algorithm, ACM Trans Math Softw 2 (1976), 378–387.

[13] F. Glover and M. Laguna, Tabu search, Kluwer Academic Publishers, Boston, 1997.

[14] G. Guan and K.L. Williams, Profile minimization of triangulated triangles, Discrete Math 260 (2003), 69–76.

[15] R. Karp, Mapping the genome: Some combinatorial problems arising in molecular biology, STOC'93 Proc Twenty-Fifth Ann ACM Symp Theory Comput, 1993, pp. 278–285.

[16] D. Kendall, Incidence matrices, interval graphs and seriation in archaeology, Pac J Math 28 (1969), 565–570.

[17] M. Laguna and R. Martí, Scatter search: Methodology and implementations in C, Kluwer Academic Publisher, Boston, 2003.

[18] J. Lewis, The Gibbs-Poole-Stockmeyer and Gibbs-King algorithms for reordering sparse matrices, ACM Transactions on Mathematical Software 8 (1982), 190–194.

[19] R. Lewis, Simulated annealing for profile and fill reduction, Int J Numer Methods Eng 37 (1994), 905–925.

[20] Y. Lin and J. Yuan, Profile minimization problem for matrices and graphs, Acta Math Appl Sinica, English-Series 10 (1994), 107–112.

[21] R. Martí, A. Duarte, and M. Laguna, Advanced scatter search for the max-cut problem, INFORMS J Comput 21 (2009), 26–38.

[22] Matrix Market 2011, webpage. http://math.nist.gov/MatrixMarket/collections/hb.html

[23] L. Michel and P. van Hentenryck, Localizer, Constraints 5 (2000), 43–84.

[24] J.J. Pantrigo, R. Martí, A. Duarte, and E.G. Pardo, Scatter search for the cut width minimization problem, Ann Oper Res 199 (2012), 285–304.

[25] M. Penrose, J. Petit, M. Serna, and J. Diaz, Convergence theorems for some layout measures on random lattice and random geometric graphs, J Combin Probab Comput 9 (2000), 489–511.

[26] J. Petit, M. Serna, and J. Díaz, A survey of graph layout problems, ACM Comput Surv 34 (2002), 313–356.

[27] W. Poole, P. Stockmeyer, and N. Gibbs, An algorithm for reducing the bandwidth and profile of a sparse matrix, SIAM J Numer Anal 13 (1976), 236–250.

[28] M.G.C. Resende, R. Martí, M. Gallego, and A. Duarte, GRASP and path relinking for the max-min diversity problem, Comput Oper Res 37 (2010), 498–508.

[29] M.G.C. Resende, S.H. Smith, and T.A. Feo, A greedy randomized adaptive search procedure for maximum independent set, Oper Res 42 (1994), 860–878.

[30] Y. Saad, Iterative methods for sparse linear systems, Society for Industrial and Applied Mathematics, San Francisco (California), 2003.

[31] R. Tewarson, Sparse matrices, Academic Press, New York, 1973.