Computers and Operations Research 87 (2017) 83–97

Contents lists available at ScienceDirect

Computers and Operations Research

journal homepage: www.elsevier.com/locate/cor

A variable neighborhood search and simulated annealing hybrid for the profile minimization problem

Gintaras Palubeckis*

Faculty of Informatics, Kaunas University of Technology, Studentu 50–408, 51368 Kaunas, Lithuania

ARTICLE INFO

Article history:
Received 10 January 2017
Revised 25 May 2017
Accepted 2 June 2017
Available online 3 June 2017

Keywords:
Combinatorial optimization
Matrix profile
Graph profile
Simulated annealing
Variable neighborhood search

ABSTRACT

Given an undirected simple graph G, the profile minimization problem (PMP) is to find an ordering of the vertices of the graph G such that the sum of the profiles of all its vertices is minimized. The profile of the vertex v in position i is defined as $\max\{0, i - h_v\}$, where h_v is the position of the leftmost vertex among all vertices adjacent to v in G. We propose an approach for the PMP, which combines a variable neighborhood search (VNS) scheme with the multi-start simulated annealing (MSA) technique. The solution delivered by MSA is submitted as input to the VNS component of the method. The VNS algorithm heavily relies on a fast insertion neighborhood exploration procedure. We show that the time complexity of this procedure is $O(n^2)$, where n is the order of G. We have found empirically that it is advantageous to give between 50 and 75% of the computation time to MSA and the rest to VNS. The results of the computational experiments demonstrate the superiority of our MSA-VNS algorithm over the current state-of-the-art metaheuristic approaches for the PMP. Using MSA-VNS, we improved the best known solutions for 50 well-recognized benchmark PMP instances in the literature. The source code implementing MSA-VNS is made publicly available as a benchmark for future comparisons.

1. Introduction

The *profile minimization problem* (PMP) is an important member of a wide set of combinatorial optimization problems on permutations. It can be stated as follows. Suppose that we are given an undirected simple graph $G = (V, E)$ with vertex set V and edge set E and a permutation of its vertices $p = (p(1), \ldots, p(n))$, where $n = |V|$ is the order of G and $p(i)$, $i \in \{1, \ldots, n\}$, is the vertex in the ith position of the permutation. We denote by $h_{p(i)}$ the leftmost position $j < i$ such that $(p(i), p(j)) \in E$. If no such j exists, then $h_{p(i)}$ is set to i. The *profile* of vertex $p(i)$ is defined as the difference between i and $h_{p(i)}$. With these notations, the PMP can be expressed as

$$\min_{p \in \Pi} F(p) = \sum_{i=1}^{n} (i - h_{p(i)}) \tag{1}$$

where Π is the set of all permutations of $V = \{1, \ldots, n\}$. The largest of the vertex profiles

$$\varphi(p) = \max_{1 \leqslant i \leqslant n} (i - h_{p(i)}) \tag{2}$$

is called the *bandwidth* of the graph G. An example of an instance of the PMP is shown in Figure 1. Historically, the PMP has received the greatest attention in the development of methods for solving sparse systems of linear equations of the form

$$Ax = b, \tag{3}$$

where $A = (a_{ij})$ is a symmetric $n \times n$ matrix, b is an n-vector, and x is the n-vector of unknowns. The system (3) is represented by the graph G in which a pair of vertices i and j are joined by an edge if and only if $a_{ij} = a_{ji} \neq 0$. The problem of solving (3) arises in various contexts in science and engineering, especially in situations where finite-element analysis is used. In many applications, the matrix A is sparse. It is well known that the direct methods for solving (3) perform better in terms of execution time when nonzero entries of the sparse matrix A are grouped around the main diagonal. A possible way to obtain such a matrix is to symmetrically permute the rows and columns of A using a good solution p to the PMP instance defined by A. For a thorough discussion on the applicability of profile minimization techniques to the solution of a system of linear equations, the reader is referred, for example, to the book of Tewarson (1973) and to the recent survey article by Davis et al. (2016). A number of other applications of the PMP have been identified in the literature. These include an approach by Xu et al. (2013) for sparse matrix-vector multiplication optimizations using graphics processing units. The au-

* Corresponding author.
E-mail address: gintaras.palubeckis@ktu.lt

http://dx.doi.org/10.1016/j.cor.2017.06.002
0305-0548/

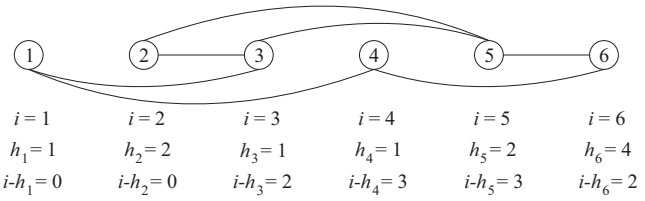

Fig. 1. Solution $p = (1, \ldots, 6)$ to an instance of the PMP in which $h_{p(1)} = h_1 = 1$, $h_2 = 2$, $h_3 = 1$, $h_4 = 1$, $h_5 = 2$, $h_6 = 4$ and $F(p) = 0 + 0 + 2 + 3 + 3 + 2 = 10$.

thors have shown that, with matrix bandwidth/profile minimization techniques, both cache usage enhancement and index compression can be enabled. Higham (2003) considered the PMP in the context of small world networks. He investigated the use of profile minimization algorithms for so-called small world reordering problem. Berry M. et al. (1996) dealt with the PMP in the information retrieval setting. They used the reordering algorithms to produce narrow-banded hypertext matrices for cluster identification. Mueller et al. (2007) reported on an application of profile reduction techniques in the graph visualization domain. The authors used graph ordering heuristics for visual analysis of data sets represented by visual similarity matrices. Meijer and de Pol (2015) suggested application of profile minimization algorithms in symbolic model checking. They addressed the problem of static ordering of variables in decision diagrams representing formulas. Bolanos M. et al. (2012) considered the PMP in relation to a new approach for computing entropy rate for undirected graphs. The matrix bandwidth and profile reduction techniques can be used to obtain lower bounds for graph entropy.

Lin and Yuan (1994) have shown the PMP to be NP-hard. Therefore, polynomial-time exact algorithms for this problem have been proposed only for restricted classes of graphs, e.g., wheels, complete bipartite graphs (Lin and Yuan, 1994), trees (Kuo and Chang, 1994), and triangulated triangles (Guan and Williams, 2003).

Because of the hardness of the problem, the majority of the research on the PMP has focused on designing heuristic and metaheuristic-based algorithms. One of the most widely studied ways of obtaining a satisfactory solution to the problem has been the use of constructive methods. They generate permutations by starting from an empty solution and gradually assigning the vertices of the graph to free positions of the permutation. One of the first constructive algorithms for the PMP was proposed by Cuthill and McKee (1969). Their algorithm generates the level structure rooted at a vertex of minimum degree. The vertices are assigned labels (in other words, the vertices are placed in the permutation) in increasing level order. George (1971) observed that reversing the obtained permutation yielded better solutions. In the literature, this approach is referred to as the Reverse Cuthill-McKee algorithm (RCM). Gibbs et al. (1976) presented another constructive technique for reducing the profile of a sparse matrix. This heuristic first finds the endpoints of a pseudo-diameter. Then it constructs a level structure and finally applies a fast numbering procedure similar to that used in RCM. At about the same time, Gibbs (1976) proposed a variation of this heuristic, which is currently known as the Gibbs-King algorithm. Sloan (1986) developed an algorithm consisting of two distinct stages. In the first stage, a pair of pseudo-peripheral vertices are located. They serve as start and end vertices for the second (numbering) stage of the method. Other vertices are chosen according to a priority function composed of two terms, one of which tries to reduce the increase in the profile and another takes into account the distance between the considered vertex and the end vertex. The algorithm was shown to be superior to previous methods (Sloan, 1986). Several enhancements to the Sloan algorithm have also been proposed (Duff et al., 1989; Kumfert and Pothen, 1997; Reid and Scott, 1999). Barnard et al. (1995) and Paulino et al. (1994a); 1994b) developed algorithms for the PMP which are based on spectral properties of the adjacency matrix of the graph. In these algorithms, the vertices are ordered according to the eigenvector corresponding to the smallest positive eigenvalue of the Laplacian matrix associated with the given graph. Hu and Scott (2001) presented a multilevel algorithm for profile reduction. Their approach combines a graph coarsening technique with the Sloan algorithm on the coarsest graph.

In the literature, there have been several local search or metaheuristic-based algorithms proposed for solving the PMP. The earliest such algorithm appeared in 1985 by Armstrong (1985). This algorithm is based on the simulated annealing (SA) paradigm. The move type that is applied in this implementation of SA is an interchange of two randomly selected vertices. The initial temperature is chosen such that almost any interchange is likely to be accepted. The algorithm incorporates a reheating mechanism which is triggered when the system has been cooled too quickly during the previous iterations of SA. Another SA algorithm for the PMP has been proposed by Lewis (1994). Like SA approach of Armstrong, it proceeds by performing random pairwise interchanges of vertices. The initial temperature depends on the profile of the starting solution. Throughout cooling, the temperature is lowered by a factor of 0.9 or 0.95. At each iteration, the algorithm computes the change in profile as a result of performing a move. Computational experience with this algorithm has been reported for several SA schedules. Hager (2002) developed two exchange methods for improving a given solution to the PMP. One of them, called down exchange, involves iteratively shifting a vertex to the right by one position at a time. Another exchange method, called up exchange, explores the solutions which can be obtained by shifting a vertex to the left in a similar manner. When combined together, these methods essentially can be categorized as a local search (LS) algorithm. The neighborhood of a solution in this approach consists of all those solutions that can be obtained by relocating a vertex from its current position in the permutation to a different one. Reid and Scott (2002) presented a significantly improved implementation of the Hager's exchange methods. An especially good performance, in terms of running time, was provided by the developed version of the up exchange algorithm. Kaveh and Sharafi (2012) proposed an algorithm for the PMP, which is based on the metaheuristic optimization method known as charged system search. Numerical experiments have demonstrated the effectiveness of this algorithm. Koohestani and Poli (2014) presented a hyper-heuristic approach based on genetic programming for evolving an enhanced version of the Sloan algorithm. The authors combined this version with the local search technique. Each step in their implementation of local search consists of swapping positions of two vertices. The approach was shown to outperform six existing algorithms for the PMP. More recently, Koohestani and Poli (2015) developed a genetic programming system for profile reduction of sparse matrices. They tested this method against several state-of-the-art heuristic techniques. Sánchez-Oro et al. (2015) proposed a scatter search algorithm for solving the PMP. They considered two solution improvement methods. One of them relies on performing pairwise interchanges of vertices. Another method involves the use of vertex insertion moves. The authors have found that the latter method was much faster than the former one. Sánchez-Oro et al. reported the results of extensive computational experiments. Their algorithm improved best known solutions for a number of benchmark PMP instances.

In some applications, e.g., in direct sparse matrix methods, it is important that a satisfactory solution to the PMP be provided very quickly. In this respect, metaheuristic-based methods are inferior to the fast heuristic techniques. However, considering direct matrix methods, Lewis (1994) as well as Reid and Scott (2002) have

noticed that local search and metaheuristic approaches are useful in situations where either the same matrix A is used repeatedly or there are a large number of matrices having the same sparsity pattern (and represented by the same graph). Moreover, such approaches are applicable in the context of validating fast heuristic procedures for profile minimization. Koohestani and Poli (2015) have remarked that an algorithm for the PMP could be used independently of direct methods like Gaussian elimination. In this case, the algorithm is not required to meet a real-time deadline. There are some other applications of the PMP where the computation time is not a major factor, and the development of metaheuristic algorithms for such applications is of significant practical as well as theoretical interest (Meijer and de Pol, 2015; Mueller et al., 2007).

For a more detailed overview of profile minimization, there are several recent surveys available. The reader is referred to Mafteiu-Scai LO (2014),Bernardes J. .A and de Oliveira SL (2015), and Davis et al. (2016).

The PMP is related to the bandwidth minimization problem (BMP), which can be expressed as $\min_{p \in \Pi} \varphi(p)$, where $\varphi(p)$ is given by (2). To deal with this problem, many metaheuristic-based solution techniques have been proposed. Recent techniques include variable neighborhood search (Mladenovic et al., 2010), tabu search (Campos et al., 2011), scatter search (Campos et al., 2011), learning based evolutionary approach (Isazadeh et al., 2012), genetic algorithm (Pop et al., 2014), and simulated annealing (Torres-Jimenez et al., 2015). A detailed description of the contributions on the BMP can be found in the above cited references.

The purpose of the present research is to offer a new algorithm for graph (matrix) profile minimization. We were motivated by the fact that very few metaheuristic-based methods for the PMP have been proposed in the literature. Our primary focus was on developing an algorithm capable of producing solutions of high quality. The algorithm presented in this paper combines a variable neighborhood search (VNS) scheme with the simulated annealing method. Our choice of VNS and SA was inspired by the great success of these techniques for some other optimization problems on permutations, e.g., the BMP on graphs (Mladenovic et al., 2010; Torres-Jimenez et al., 2015) and the single row facility layout problem (Palubeckis, 2015; 2017). The role of the SA component is to generate a good initial solution for VNS. We have implemented this component as a multi-start simulated annealing (MSA) algorithm. The VNS component of the approach tries to improve the currently best solution by repetitively applying a local search (LS) procedure. As a termination condition, we use a maximum CPU time limit for a run. An interesting question is how to divide up this limit into two time limits, one for MSA and another for VNS. We report empirical results related to this question. As mentioned before, the VNS component employs a LS procedure. As a byproduct of our approach, we present such a procedure for the case when an insertion neighborhood structure is used. It is clear that for an n-vertex graph there are $n(n-1)$ insertion moves possible. Using the expression in the right-hand side of (1), the gain of an insertion move can be computed in $O(n^2)$ operations. Thus the complexity of this straightforward approach for exploration of the entire neighborhood is $O(n^4)$. In this paper, we propose an insertion neighborhood exploration procedure running in $O(n^2)$ time. This procedure is very fast, requiring only $O(1)$ operations per move. Actually, it has (asymptotically) the same time complexity as the one-time evaluation of the objective function F.

The remainder of this paper is arranged as follows. In the next section, we present the hybrid MSA-VNS algorithm for the PMP. In Section 3, we propose a local search procedure used by our VNS implementation. In Section 4, we report computational experience on a set of benchmark PMP instances. Concluding remarks are given in Section 5.

```
    MSA-VNS
1:  if θ > 0 then
2:     Apply MSA algorithm to get an initial permutation p
3:     if θ = 1 then Stop with the solution p of value F(p)
4:  else
5:     Generate an initial permutation p at random
6:  end if
7:  Apply VNS(p, p*, f*)
8:  Stop with the solution p* of value f*
```

Algorithm 1. Variable neighborhood search combined with multi-start simulated annealing.

2. The algorithm

Variable neighborhood search (Hansen and Mladenović, 2001; Hansen et al., 2010) is a metaheuristic developed for the solution of optimization problems. This general-purpose optimization method combines a neighborhood change mechanism with a local search technique. In our implementation of VNS for the PMP, we rely on the neighborhood structures that are quite natural for problems whose solution space is composed of permutations. Specifically, let $p \in \Pi$ be fixed. Then the ith neighborhood $N_i(p)$ of p consists of all permutations $p' \in \Pi$ that differ from p on exactly i entries. Our algorithm makes use of a subset of the system of neighborhoods $N_2, N_4, N_6, \ldots, N_{2\lfloor n/2 \rfloor}$. A permutation $p' \in N_{2k}(p)$, $k \in \{1, \ldots, \lfloor n/2 \rfloor\}$, is generated from p by performing k pairwise interchanges of vertices under the restriction that no vertex may be moved to a new position in the permutation more than once. An initial solution for VNS is either provided by the MSA algorithm or generated randomly. This functionality is controlled by the parameter θ. The value of this parameter stands for the fraction of time limit used for simulated annealing. To make this more precise, suppose the maximum CPU time limit is T_{lim} time units. Then θT_{lim} time units are allocated for running MSA and the remaining $(1-\theta)T_{\text{lim}}$ time units are reserved for executing VNS. Certainly, the parameter θ is restricted to be between 0 and 1. If $\theta = 0$, then the initial solution for VNS is a random permutation. A different situation is when $\theta = 1$. In this case, only MSA is invoked, and the solution produced by MSA is the output of the algorithm. If θ lies within open interval (0, 1), then both MSA and VNS are executed.

The pseudo-code of the top level of the MSA-VNS algorithm for the PMP is presented in Algorithm 1. As mentioned before, if $0 < \theta < 1$, an initial solution for VNS is provided by the MSA method. Its pseudo-code is shown in Algorithm 2. Simulated annealing is a metaheuristic method that has been widely used to find high quality solutions for various optimization problems. The basic element of the search mechanism in this method is the acceptance probability (Černý, 1985; Kirkpatrick et al., 1983) $\exp(-(F(p') - F(p))/T)$, where p is the current permutation, p' is a permutation in a neighborhood of p, and T is the temperature value. The difference $\delta(p, p') = F(p') - F(p)$ is called a move gain. We implemented SA (and MSA) for the case where p' belongs to the pairwise interchange neighborhood $N_2(p)$. As it is well known, SA starts with T set to a high value, denoted by $T_{\max}$, and terminates when T reaches a value, denoted by $T_{\min}$, which is very close to zero. We choose $T_{\min} = 0.0001$ and set $T_{\max}$ to $\max_{p' \in N'} |F(p') - F(p)|$, where $p \in \Pi$ is a random permutation (generated in line 1 of Algorithm 2) and N' is a randomly selected subset of $N_2(p)$. The size of N' is fixed at 5000. The temperature is decreased according to the geometric schedule by setting $T := \alpha T$, where α is a cooling factor whose value usually lies in the interval [0.9, 0.995]. We fixed α at 0.95. Another parameter of SA is the number of moves, m, to be attempted at each temperature level. In the literature (see, for example, Rutenbar, 1989), this number often is chosen to be $100n$. We set this number to $100n$, too. Our

```
      MSA
// Input to MSA includes parameters α, m and T_min
 1:  Randomly generate a permutation p ∈ Π and initialize h_v, L_v, v ∈ V
 2:  p* := p
 3:  Compute T_max and τ̄ = ⌊(log(T_min) − log(T_max))/ log α⌋
 4:  while maximum time limit for MSA not reached do
 5:    f := F(p)
 6:    T := T_max
 7:    for τ = 1, ..., τ̄ do
 8:      for i = 1, ..., m do
 9:        Pick two vertices r and s at random
10:        δ:=get_gain(r, s, k, l) // k and l are positions of r and s, respectively
11:        if δ ⩽ 0 or exp(−δ/T) ⩾ random(0, 1) then
12:          f := f + δ
13:          Swap positions of the vertices r and s and update h_v, L_v, v ∈ V
14:          if f < F(p*) then p* := p end if
15:        end if
16:      end for
17:      T := αT
18:    end for
19:    Randomly generate a permutation p ∈ Π and initialize h_v, L_v, v ∈ V
20:  end while
21:  return p*
```

Algorithm 2. Multi-start simulated annealing.

```
      get_gain(r, s, k, l) // assume w.l.o.g. that k < l
 1:  δ := 0
 2:  for i = k + 1, ..., l − 1 do
 3:    for each u ∈ L_{p(i)} do
 4:      if u ≠ s and (u, s) ∈ E then δ := δ + (i − k) end if
 5:    end for
 6:    if (p(i), s) ∈ E and h_{p(i)} = i then δ := δ + (i − k) end if
 7:  end for
 8:  δ := δ + (l − k)|L_s|
 9:  δ := δ − (l − max(k, h_s))
10:  for each u ∈ L_r \ {s} such that (u, s) ∉ E do
11:    if B'_u := {w ∈ Q_u | k < π(w) < π(u)} = ∅ then
12:      δ := δ − (min(l, π(u)) − k)
13:    else
14:      Find the leftmost vertex in B'_u (let it be denoted by v)
15:      δ := δ − (min(l, π(v)) − k)
16:    end if
17:  end for
18:  if h_r < k then δ := δ + (l − k)
19:  else
20:    q := n
21:    for w ∈ Q_r such that π(w) ⩽ l do
22:      if π(w) = l then q := k
23:      else if π(w) < q then q := π(w) end if
24:      end if
25:    end for
26:    if q < n then δ := δ + (l − q) end if
27:  end if
28:  return δ
```

Algorithm 3. Computing the move gain.

implementation of the SA method is a multi-start algorithm, MSA, in which starting solutions for SA are generated randomly (lines 1 and 19). An important feature of the approach is that the parameters of MSA are retained unchanged from one restart to the next. In the pseudo-code, p^* denotes the best found solution and L_v, $v \in V$, are subsets of V used by the procedure `get_gain` (they will be defined later in this section).

Various implementations of SA differ in the way how the move gain is computed. The pseudo-code of our procedure for computing the move gain $\delta(p, p')$, named `get_gain`, is depicted in Algorithm 3. The permutation $p' \in N_2(p)$ is obtained by swapping positions of the vertices $r = p(k)$ and $s = p(l)$ in the permutation p. The procedure uses two types of sets. For $v \in V$, the set Q_v consists of all vertices adjacent to v. Another set is $L_v = \{u \in V \setminus \{v\} \mid h_u = \pi(v)\}$, where π is the inverse permutation of p, defined as follows: if $p(i) = v$, then $\pi(v) = i$. Clearly, L_v is a subset of Q_v. In Figure 1, $L_1 = \{3, 4\}$, $L_2 = \{5\}$, $L_4 = \{6\}$, $L_3 = L_5 = L_6 = \emptyset$. The procedure goes through three stages. In the first stage (lines 2–7), each vertex $u \in L_{p(i)} \setminus \{s\}$ with $i \in K := \{k+1, k+2, \ldots, l-1\}$ is examined. If u is adjacent to s, then, after interchanging r and s, the profile of u increases by $i - k$. For example, in Figure 2, $k = 1$, $l = 5$ and, for $i = 2$, $u = 3 \in L_{p(i)} = L_2$, $(u, s) = (3, 5) \in E$ and therefore δ is set to $\delta + (i - k) = 0 + (2 - 1) = 1$. It may happen that $h_{p(i)} = i$ holds for the vertex $p(i)$, $i \in K$ (then $p(i)$ does not belong to any $L_{p(j)}$, $j \in K$, and, consequently, is never taken as u in the loop given in lines 3–5). In this case, if $p(i)$ is adjacent to s, then the gain value δ is increased too (line 6). In Figure 2, such a vertex is $p(2) = 2$. Therefore, δ is set to $\delta + (i - k) = 1 + (2 - 1) = 2$. In the second stage (lines 8, 9), the vertex s is processed. Because s is moved to position k in p, the profile of each vertex $u \in L_s$ increases by $l - k$, and the profile of s decreases by $l - \max(k, h_s)$. Notice that h_s may be smaller than k. In Figure 2, $h_s = 2$ and δ is set to $2 - (5 - \max(1, 2)) = -1$. Finally, in the third stage (lines 10–27), the vertices in the set $L_r \setminus \{s\} \cup \{r\}$ are considered. For each vertex $u \in L_r \setminus \{s\}$ non-adjacent to s, the set B'_u is formed by including all the vertices adjacent to u which are positioned to the right of r and to the left of u in the permutation p. If this set is empty, then the gain δ is decreased by $l - k$ if the vertex u lies to the right of s, and by $\pi(u) - k$ otherwise. In our example, B'_6 is empty for the vertex $6 \in L_r = L_1$ (see Figure 2). Therefore, δ is set to $\delta - (l - k) = -1 - (5 - 1) = -5$. If B'_u is nonempty, then the procedure finds the leftmost vertex v in this set. The profile of the vertex u decreases by $\min(l, \pi(v)) - k$, and the gain is updated accordingly. In Figure 2, $B'_4 = \{3\}$ for the vertex $4 \in L_1$, and thus δ is set to $\delta - (\pi(3) - k) = -5 - (3 - 1) = -7$. In the second part of the third stage (lines 18–27), the vertex r is considered. Clearly, if $h_r < k$, then the profile of r increases by $l - k$. Otherwise ($h_r = k$), `get_gain` proceeds by identifying the leftmost vertex, say v, among all vertices that are adjacent to r and are placed to the left of $p(l + 1)$. While performing this operation, it is assumed that the vertex s is moved to position k in p. If the vertex being searched for is found ($q < n$ in the pseudo-code), then the gain δ is increased by $l - \pi(v) = l - q$. In Figure 2, $q = \pi(4) = 4$,

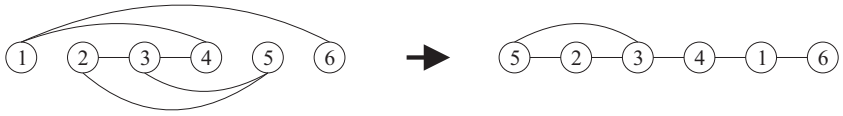

Fig. 2. Swapping positions of vertices 1 and 5.

```
    VNS(p, p*, f*)
1:  p* := p
2:  f* := F(p)
3:  Initialize arrays Y and Z with zeros
4:  f :=local_search(p, π, Y, Z) // constructs a locally optimal solution p
5:  if f < f* then
6:    p* := p
7:    f* := f
8:  end if
9:  while maximum time limit for VNS not reached do
10:   k := k_min
11:   Compute k_max and k_step
12:   while k ⩽ k_max do
13:     shake(p, p*, k)
14:     f :=local_search(p, π, Y, Z)
15:     neighborhood_change(p, p*, f, f*, k, k_min, k_step)
16:   end while
17: end while
18: Return solution p* of value f*
```

Algorithm 4. Variable neighborhood search.

```
    shake(p, p*, k)
1:  p := p*
2:  W := {1, ..., n}
3:  for k times do
4:    Randomly select vertices u, v ∈ W
5:    Interchange positions of u and v in p
6:    W := W \ {u, v}
7:  end for
```

Algorithm 5. Shake function.

```
    neighborhood_change(p, p*, f, f*, k, k_min, k_step)
1:  if f < f* then
2:    p* := p
3:    f* := f
4:    k := k_min
5:  else
6:    k := k + k_step
7:  end if
```

Algorithm 6. Neighborhood change function.

and the value of δ becomes equal to $-7+(5-4)=-6$. For the left and right solutions in Figure 2, $F=0+0+1+3+3+5=12$ and, respectively, $F=0+1+2+1+1+1=6=12+\delta$.

It can be seen from Algorithm 3 that the first stage is the most time consuming part of `get_gain`. The worst-case time complexity of this part, and hence of the entire procedure, is $O(nd_{\max})$, where $d_{\max}$ is the maximum vertex degree of the graph G.

The pseudo-code of the VNS component of the approach is shown in Algorithm 4. Starting from an initial permutation p, VNS first initializes the best found solution p^*, its objective value f^* and arrays Y and Z. The arrays Y and Z are needed for an efficient implementation of the local search procedure. They will be defined in the next section, where this procedure will be described in detail. After initialization, the algorithm proceeds with a call to `local_search` (line 4 of VNS). The parameter π stands for the inverse permutation of p as before. If the solution returned by `local_search` is better than the initial one, then p^* and f^* are updated (lines 5–8). The main body of VNS consists of two nested "while" loops. To stop the outer loop, we use a maximum CPU time limit, $(1-\theta)T_{\text{lim}}$, as a termination condition. Before entering the inner loop, the algorithm selects the value of $k_{\max}$, which defines the largest possible size of the neighborhood in the search process, and the value of k_{step}, which is used to move from the current neighborhood to the next one. The value of $k_{\max}$ is an integer number drawn uniformly at random from the interval $[\kappa_1 n, \kappa_2 n]$. We have fixed κ_1 at 0.1 and κ_2 at 0.4. This choice is guided by the need to avoid almost random restarts (when κ_2 is close to 0.5) and do not restrict the search to exploring only relatively small neighborhoods (when $\kappa_1 < 0.1$). Once $k_{\max}$ is selected, the value of k_{step} is set to $\max(\lfloor k_{\max}/\lambda \rfloor, 1)$, where λ is a parameter of the algorithm. Another parameter of VNS is $k_{\min}$, which specifies the size of the neighborhood the search is started from. The inner "while" loop performs a number of iterations. At the beginning of each iteration, the algorithm applies a shaking procedure (see Algorithm 5) to the best permutation, p^*, found thus far. The perturbed solution belongs to the neighborhood $N_{2k}(p^*)$. This solution is improved by making a call to `local_search` procedure (line 14 of VNS). An iteration ends with a few statements implementing a neighborhood change function (see Algorithm 6).

As it can be observed in Algorithms 4 and 5, each iteration of VNS uses the best solution p^* as a starting point. One may guess that if initially (line 1 of VNS) this solution is not far from the optimum, then the algorithm is able to provide faster convergence and find a final solution of better quality. Our computational experience confirms that for the PMP, this is indeed the case. We tested two scenarios. In the first of them, we generated an initial permutation p at random and improved it by `local_search`. In the second case, we applied `local_search` to a solution delivered by the multi-start simulated annealing algorithm. In this case, typically the initial value of f^* (right before line 9) was smaller and the VNS algorithm performed significantly better than in the first scenario.

3. Local search

The local search procedure is at the heart of the developed VNS algorithm for the PMP. Like in Hager (2002); Reid and Scott (2002); Sánchez-Oro et al. (2015), our LS implementation is based on the exploration of the insertion neighborhood. We denote this neighborhood structure by $\tilde{N}(p)$, $p \in \Pi$. The set $\tilde{N}(p)$ consists of all permutations that can be obtained from p by removing a vertex from its current position in p and inserting it at a different position. Formally, $\tilde{N}(p) = \{p' \in \Pi \mid$ there exist k, $l \in I$, $k \neq l$, such that $p'(l) = p(k)$, $p'(i) = p(i-1)$, $i = l+1, \ldots, k$, if $l < k$, $p'(i) = p(i+1)$, $i = k, \ldots, l-1$, if $l > k$, and $p'(i) = p(i)$ for the remaining $i \in I\}$, where $I = \{1, \ldots, n\}$. In our LS algorithm, we adopt the best improvement strategy applied to a subset of the neighborhood $\tilde{N}(p)$ of the current solution p, which means that this subset is explored exhaustively and the best neighbor of p in it is identified. The details on how the search is reduced are provided later in this section.

As alluded to before, the algorithm uses two auxiliary arrays, Y and Z. They are related to $n \times n$ matrices $\tilde{Y} = (\tilde{y}_{uv})$ and, respectively, $\tilde{Z} = (\tilde{z}_{vu})$ with entries defined as follows. For a vertex u and a vertex v positioned to the right of u, $\tilde{y}_{uv} = |L_u \cap Q_v|$. Other entries of the matrix $\tilde{Y}$ are equal to zero by definition. In part (a) of Figure 3, $\tilde{y}_{13} = |L_1 \cap Q_3| = 2$, $\tilde{y}_{23} = |L_2 \cap Q_3| = 0$, $\tilde{y}_{24} =$

Fig. 3. Graphs used to illustrate the definition of the matrices $\tilde{Y}$ (part (a)) and $\tilde{Z}$ (part (b)).

```
    local_search(p, π, Y, Z)
1:  Form sets L_v and B_v for each vertex v ∈ V and initialize vector H = (h_1, ..., h_n)
2:  Compute the objective function value f := F(p)
3:  Δ* := −1
4:  while Δ* < 0 do
5:    Δ*:=explore_neighborhood(p, π, Y, Z)
6:    if Δ* < 0 then f := f + Δ* end if
7:  end while
8:  return f
```

Algorithm 7. Main procedure of LS.

```
    explore_neighborhood(p, π, Y, Z)
1:  Δ* := 0
2:  for each v ∈ V do
3:    fill_arrays(v, p, π, Y, Z)
4:    explore_left(v, p, π, Δ*, v*, l*, Y)
5:    explore_right(v, p, π, Δ*, v*, l*, Z)
6:    clear_arrays(v, p, π, Y, Z)
7:  end for
8:  if Δ* < 0 then
9:    k := π(v*)
10:   Relocate vertex v* from position k to position l* in p
11:   Update the inverse permutation π correspondingly
12:   if l* < k then update_left(v*, k, l*, p)
13:   else update_right(v*, k, l*, p, π) end if
14: end if
15: return Δ*
```

Algorithm 8. Neighborhood exploration procedure.

```
    fill_arrays(v, p, π, Y, Z)
1:  for each w ∈ Q_v do
2:    u := p(h_w)
3:    if u ≠ w and π(u) < π(v) then y_u := y_u + 1 end if
4:  end for
5:  for each w ∈ L_v do
6:    i_min := n
7:    for each s ∈ Q_w do
8:      if s ≠ v, π(s) < π(w) and π(s) < i_min then
9:        i_min := π(s)
10:       u := s
11:     end if
12:   end for
13:   if i_min < n then z_u := z_u + 1 else z_w := z_w + 1 end if
14: end for
```

Algorithm 9. Forming arrays Y and Z.

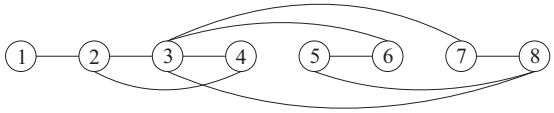

Fig. 4. Permutation used to illustrate procedure `fill_arrays` (applied to the vertex $v = 3$).

$|L_2 \cap Q_4| = 1$, $\tilde{y}_{15} = |L_1 \cap Q_5| = 2$, and $\tilde{y}_{46} = |L_4 \cap Q_6| = 1$. To define $\tilde{Z}$, consider vertices v, w such that $w \in L_v$. Let us denote by ρ_w the leftmost vertex in $B_w \cup \{w\} \setminus \{v\}$ where B_w is the set of all vertices adjacent to w that appear to the left of w in the permutation p. Then, for a vertex v and a vertex u placed to the right of v, $\tilde{z}_{vu} = |\{w \in L_v \mid \rho_w = u\}|$. Other entries of the matrix $\tilde{Z}$ are assumed to be zero. In the example given in Figure 3, part (b), $\rho_5 = 2$ with respect to $v = 1$, and $\rho_3 = \rho_4 = \rho_6 = 3$ with respect to $v = 2$. Therefore, the only nonzero entries of $\tilde{Z}$ are $\tilde{z}_{12} = 1$ and $\tilde{z}_{23} = 3$. In our LS algorithm, the array Y is reused for storing the columns of the matrix $\tilde{Y}$, and the array Z is reused for storing the rows of the matrix $\tilde{Z}$. Notice that all these data structures are dependent on the permutation p. However, in order to lighten notations, we skip writing this dependency explicitly.

We now describe the insertion-based LS algorithm for the PMP. For better readability, we present it in a top-down manner. The main routine of the algorithm is shown in Algorithm 7. It consists of an initialization phase followed by a loop which repeatedly calls the procedure `explore_neighborhood`. The pseudo-code of the latter is given in Algorithm 8. Since the best improvement search strategy is applied, `explore_neighborhood` examines all the solutions in a certain subset of the neighborhood $\tilde{N}(p)$ and selects the one for which the minimum of the function F is attained. The selected move is specified by the vertex v^* and its new position l^* in p. The best gain value is denoted by Δ^*. Negative Δ^* means that an improving move has been found. In this case, the updated permutations p and π are returned. For each vertex $v \in V$, `explore_neighborhood` first makes a call to the procedure `fill_arrays` (Algorithm 9). The first loop of the procedure (lines 1–4) forms the column of the matrix $\tilde{Y}$ corresponding to the vertex v. The entries of this column are stored in the array Y. Notice that if the profile of the vertex w is positive (or, equivalently, $u \neq w$), then $w \in L_u$. Since also $w \in Q_v$, the value of $y_u = \tilde{y}_{uv}$ is incremented by 1. The execution of the procedure is illustrated using the permutation shown in Figure 4. Suppose that $v = 3$. Then $Q_v = \{2, 4, 6, 7, 8\}$. For the vertex $w = 2$, we have $h_w = 1$, $u = p(h_w) = 1$, and therefore $y_u = y_1$ is set to 1. The vertex $w = 4$ is processed similarly. In this case, $h_w = 2$, $u = 2$ and $y_u = y_2 = 1$. The remaining vertices in Q_v do not satisfy the conditions in line 3 of the pseudo-code. The rest of `fill_arrays` (lines 5–14) forms the row of the matrix $\tilde{Z}$ corresponding to v. The inner loop (lines 7–12) identifies the vertex which acts as ρ_w in the definition of the entry $\tilde{z}_{vu}$ of $\tilde{Z}$. Indeed, ρ_w is equal to u if the set $B_w \setminus \{v\}$ (a subset of Q_w defined by conditions $s \neq v$ and $\pi(s) < \pi(w)$) is nonempty and is equal to w otherwise (in this case, $i_{\min}$ stays equal to n). Depending on which case occurs, either $z_u = \tilde{z}_{vu}$ or $z_w = \tilde{z}_{vw}$ is increased by 1. To illustrate the construction of the array Z, we return to Figure 4. We continue to assume that $v = 3$ and examine the vertices in $L_v = \{6, 7, 8\}$. First suppose that $w = 6$. Then $Q_w = \{3, 5\}$, $u = 5$, and $z_u = z_5$ is set to 1. For $w = 7$, we have

```
    explore_left(v, p, π, Δ*, v*, l*, Y)
 1: Δ := 0
 2: γ := |L_v|
 3: for l = π(v) − 1 to 1 by −1 do
 4:    u := p(l)
 5:    β := y_u
 6:    Δ' := γ − |L_u| + β
 7:    if (u, v) ∈ E then
 8:       if h_v = l then Δ' := Δ' + 1 end if
 9:       if h_u = l then β := β + 1 end if
10:    else
11:       if h_v < l then Δ' := Δ' − 1 end if
12:       if h_u < l then Δ' := Δ' + 1 end if
13:    end if
14:    γ := γ + β
15:    Δ := Δ + Δ'
16:    if Δ ⩾ M then break // leave "for" loop
17:    end if
18:    if Δ < Δ* then
19:       Δ* := Δ
20:       v* := v
21:       l* := l
22:    end if
23: end for
```

Algorithm 10. Shifting the vertex v to the left.

$Q_w = \{3, 8\}$. In this case, however, $i_{\min} = n = 8$, and a value of 1 is assigned to $z_w = z_7$. Finally, consider the vertex $w = 8$. Its adjacency set $Q_w = \{3, 5, 7\}$ contains two vertices, namely 5 and 7, for which the first two conditions in line 8 are met. The choice is made in favor of 5 because this vertex is placed to the left of 7. Thus $u = 5$, and z_5 is increased from 1 to 2.

Once the arrays Y and Z are ready, `explore_neighborhood` can start examining solutions obtained from p by relocating a vertex v from the current position $\pi(v)$ to the other $n-1$ positions. First, the vertex v is shifted to the left by one position at a time. The gains resulting from these moves are computed by `explore_left` procedure shown in Algorithm 10. In the pseudo-code, the gain of the insertion move is represented by the variable Δ and its increment by Δ'. The value of Δ' is added to Δ when the vertex v is interchanged with its current left neighbor, denoted by u. The variable γ is used to represent the cardinality of the set L_v. Notice that, when v is moving to the left, new vertices can be added to this set. Specifically, after swapping positions of the vertices v and u, the current value of γ is increased either by y_u or by $y_u + 1$. Initially (line 6), Δ' is set to $\gamma - |L_u| + y_u$. This value, however, may need to be corrected. There are two cases to consider. For the first case, suppose that the vertices v and u are connected in G. If, in addition, $h_v = l$, then Δ' is incremented by 1 (line 8). This situation is depicted in Figure 5. The vertex $v = 3$ is interchanged with $u = 2$. It can be seen that $\gamma = |L_3| = 0$, $\beta = y_2 = 1$ (because (2, 4), (3, 4) $\in E$), and $|L_2| = 3$. Initially, $\Delta' = \gamma - |L_2| + \beta = -2$. Since $h_3 = l = 2$, the value of Δ' is corrected by setting $\Delta' := \Delta' + 1 = -1$. It can be checked that, for the permutation $p = (1, 2, 3, 4, 5, 6)$, $F(p) = 0 + 1 + 1 + 2 + 1 + 4 = 9$, and, after swapping positions of vertices 3 and 2 (right part of Figure 5), $F = 0 + 2 + 0 + 2 + 1 + 3 = 8$. If $h_u = l$ in line 9 of the pseudo-code, then β is incremented by 1. In Figure 5, $h_2 = 1 < l = 2$. Therefore, β is not updated, and γ is set to 1 (line 14). A different situation is shown in Figure 6, where $\gamma = |L_3| = 0$, $\beta = y_2 = 1$, $|L_2| = 2$, and $\Delta' = -1$. Since $h_2 = l = 2$, β is increased by 1 and γ is increased by 2. Meanwhile, Δ' retains the same value because $h_3 = 1 < l = 2$. The objective function values for the solutions in the left and right parts of Figure 6 are $F = 0 + 0 + 2 + 2 + 1 + 4 = 9$ and $F = 0 + 1 + 1 + 2 + 1 + 3 = 8$, respectively. For the second case, suppose that there is no edge between v and u in G. Then Δ' is decreased (respectively, increased) by 1 if $h_v < l$ (respectively, $h_u < l$) (lines 11, 12). This case is illustrated in Figure 7, where $v = 3$, $u = 2$, $\gamma = |L_3| = 1$, $\beta = y_2 = 1$, $|L_2| = 2$, and initially $\Delta' = 0$. Afterwards, Δ' is reduced to -1 because $h_v = h_3 = 1 < l = 2$ and $h_u = h_2 = l = 2$. The objective function values of the two solutions in Figure 7 are $F = 0 + 0 + 2 + 2 + 2 + 4 = 10$ and, respectively, $F = 0 + 0 + 1 + 2 + 3 + 3 = 9$.

The procedure `explore_left` includes a possibility to exit the "for" loop prematurely when the gain Δ reaches a specified positive value. This option is controlled by the parameter M. In the preliminary phase of experimentation, we found that the strategy to end the search when Δ becomes relatively large allows reducing the computation time of LS significantly and, in most cases, increasing the quality of solutions produced by VNS (because faster LS enables VNS to execute more iterations in the given time frame).

The pseudo-code of the procedure `explore_right` is presented in Algorithm 11. In the "for" loop (lines 4–22), the insertions of the vertex v to the right of $\pi(v)$ are examined. Like in `explore_left`, γ stands for the cardinality of the current set L_v. It is clear that γ can only decrease (by $z_{p(l)}$ in line 6) when v is moving to the right. In the pseudo-code, μ is a flag variable indicating whether the vertex v is adjacent to at least one vertex placed to the left of v. Provided initialized with `false` (line 3), μ is set to `true` whenever v is interchanged with a vertex $u = p(l)$ which is adjacent to v in G (line 11). The search is stopped when the condition in line 7 is met. If this is the case, then no position $k \in \{l, \ldots, n\}$ for v exists which is better than the currently best one. Indeed, since $\gamma \leq 0$ and can only decrease, the increment of the gain Δ' computed in line 9 is nonnegative during the remaining iterations of the procedure.

Figures 8 and 9 illustrate the computation of Δ' for the cases of $(u, v) \in E$ and $(u, v) \notin E$, respectively. For the interchange shown in Figure 8, $v = 1$, $|L_1| = 4$, $\mu =$`false`, $z_u = z_2 = 2$, $\gamma = |L_1| - z_2 = 2$, $|L_2| = 1$, and $\Delta' = |L_2| - \gamma = -1$. Since $(1, 2) \in E$, the flag μ in line 11 is set to `true`. By swapping positions of the vertices 1 and 2 the objective function value decreases from $F = 0 + 1 + 2 + 3 + 3 + 5 = 14$ to $F = 1 + 0 + 2 + 2 + 4 + 4 = 13$. It can be checked that in the next iteration the value of F can be decreased to 11 (indeed, $z_u = z_3 = 0$, $\gamma = 2$, $|L_3| = 0$ and, therefore, $\Delta' = |L_3| - \gamma = -2$). For the move shown in Figure 9, $v = 2$, $|L_2| = 2$, $\mu =$`false`, $z_u = z_3 = 2$, $\gamma = |L_2| - z_3 = 0$, $|L_3| = 0$, and $\Delta' = |L_3| - \gamma = 0$. Since $h_3 = 1 < l = 3$, Δ' in line 14 is reduced to -1. The move allows decreasing the objective function value from $F = 0 + 0 + 2 + 2 + 1 + 4 = 9$ to $F = 0 + 0 + 1 + 2 + 1 + 4 = 8$. Notice that this example illustrates the necessity of including $\mu =$`true` in the condition of the statement in line 7. If the simplified condition $\gamma \leq 0$ was used, then the above-considered improving move would be missed.

In line 6 of `explore_neighborhood`, the procedure `clear_arrays` is invoked. This procedure is very similar to `fill_arrays` shown in Algorithm 9. It simply sets all nonzero entries of the arrays Y and Z to 0 (instead of incrementing them as in `fill_arrays`, lines 3 and 13).

If, after termination of the loop in `explore_neighborhood`, it appears that $\Delta^* < 0$, then the selected vertex v^* is relocated from its current position $\pi(v^*)$ in p to position l^* (lines 8–10 in Algorithm 8). Obviously, this action requires updating the inverse permutation π (line 11). Depending on whether the new position of v^* is to the left or right of $\pi(v^*)$, either `update_left` or `update_right` procedure is executed. The pseudo-code of `update_left` is given in Algorithm 12. The first loop (lines 1–9) processes the vertices that have been shifted by one position to the right. In the example presented in Figure 10, such vertices are 2,3 and 4. If the vertex $u = p(i)$ is adjacent to v^*, then the sets B_u, B_{v^*} and, possibly, L_u, L_{v^*} are updated (lines 4–6). Specifically, u is removed from B_{v^*} and v^* is added to B_u. For example,

Fig. 5. Shifting the vertex $v = 3$ by one position to the left: the case of $(v, p(l)) = (3, 2) \in E$ and $h_v = h_3 = 2 = l$.

Fig. 6. Shifting the vertex $v = 3$ by one position to the left: the case of $(v, p(l)) = (3, 2) \in E$ and $h_v = h_3 = 1 < l$.

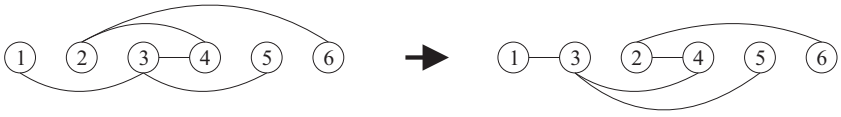

Fig. 7. Shifting the vertex $v = 3$ by one position to the left: the case of $(v, p(l)) = (3, 2) \notin E$.

```
    explore_right(v, p, π, Δ*, v*, l*, Z)
 1: Δ := 0
 2: γ := |L_v|
 3: if h_v < π(v) then μ :=true else μ :=false end if
 4: for l = π(v) + 1, ..., n do
 5:    u := p(l)
 6:    γ := γ − z_u
 7:    if γ ⩽ 0 and μ =true then break // leave "for" loop
 8:    end if
 9:    Δ' := |L_u| − γ
10:    if (u, v) ∈ E then
11:       if μ =false then μ :=true end if
12:    else
13:       if μ =true then Δ' := Δ' + 1 end if
14:       if h_u < l then Δ' := Δ' − 1 end if
15:    end if
16:    Δ := Δ + Δ'
17:    if Δ < Δ* then
18:       Δ* := Δ
19:       v* := v
20:       l* := l
21:    end if
22: end for
```

Algorithm 11. Shifting the vertex v to the right.

in Figure 10, $u = 2$ is removed from B_5 and $v^* = 5$ is added to B_2. In the condition of line 5, $i - 1$ stands for the old position of the vertex u (that is, before shifting). The statement in this line is executed only if v^* is adjacent to a shifted vertex and not adjacent to each of the preceding vertices in p. Execution of the statement in line 6 depends on whether or not h_u is less than $i - 1$. If so, then u is moved from $L_{p(h_u+1)}$ to L_{v^*}. If not, then u is simply added to L_{v^*}. For vertex $u = p(4) = 3$ in Figure 10, $h_u = l^* = 2$, $(u, v^*) \in E$ and $h_u < i - 1 = 3$. Therefore, u is moved from the set $L_{p(3)} = L_2$ to the set L_5. For vertex $u = p(3) = 2$, we have $h_u = l^* = 2 = i - 1$ and in this case u is added to L_5. In line 8, h_u is set to l^* if u is adjacent to v^* or incremented by 1 otherwise. In Figure 10, h_u is set to $l^* = 2$ for both $u = 2$ and $u = 3$ (this assignment in these particular cases, however, has no effect since $h_u = l^*$ holds already for the initial permutation p). A different situation is with $u = 4$. In this case $(u, v^*) \notin E$, and thus h_u is simply increased from 3 to 4. The second loop in Algorithm 12 (lines 10–14) iterates over all the vertices lying to the right of the previous position of v^* in the permutation p. It is clear that no changes are required for a vertex u with $h_u < l^*$ or $h_u > k$. Suppose that the condition in line 13 is fulfilled for the vertex u. If, in addition, $(u, v^*) \in E$, then h_u is set to l^* (which is the new position of the vertex v^* in p). If, however, u is not adjacent to v^*, then h_u is incremented by 1 (because the vertex in position h_u has been shifted to the right). For vertex $u = 6$ in Figure 10, $h_u = l^* = 2$ and $(u, v^*) \in E$. Therefore, the statements in lines 12 and 13 are executed: the vertex 6 is transferred from the set L_2 to the set L_5 and h_6 is set to 2 (in fact, h_6 retains the same value). In the last line, `update_left` checks the condition $h_{v^*} > l^*$, and if it holds, assigns l^* as a new value to h_{v^*}. We note that if $l^* \leqslant h_{v^*} < k$, the vertex v^* is removed from L_u, $u = p(h_{v^*} + 1)$, in line 5 of the procedure.

The pseudo-code of `update_right` is depicted in Algorithm 13 and a little example illustrating this procedure is shown in Figure 11. The loop in the pseudo-code (lines 1–9) iterates over all the vertices from $p(k)$ till the end of the permutation, with the exception of v^*. If the vertex $u = p(i)$ is adjacent to v^* and is placed to the left of v^*, then at the beginning of the iteration (line 3) the sets B_u and B_{v^*} are updated: the vertex v^* is removed from B_u and the vertex u is added to B_{v^*}. To proceed, `update_right` compares h_u with k. If $h_u \neq k$ and, moreover, $h_u \in [k + 1, \ldots, l^*]$, then h_u is decremented by 1 (for the example in Figure 11, h_4 is reduced from 3 to 2). If $h_u = k$, then three cases are possible: 1) $B_u = \emptyset$; 2) $B'_u := \{s \in B_u \mid k \leqslant \pi(s) < l^*\} = \emptyset$; 3) $B'_u \neq \emptyset$. In the first case, the vertex u is removed from L_{v^*} and h_u is set to i. In Figure 11, this is done for the vertex $u = 5 = p(4)$ (notice that B_5 has been emptied in line 3 of the pseudo-code). In the second case, h_u is set to l^*. Finally, in the third case, the leftmost vertex in p among the vertices of B'_u is selected. Let this vertex be denoted by w. The procedure moves the vertex u from the set L_{v^*} to the set L_w and assigns $\pi(w)$ to h_u. In Figure 11, these operations are performed for the vertices $u = 6$ and $w = 4$. The rest of `update_right` (lines 10–13) processes the vertex v^*. Clearly, no changes are required if $h_{v^*} < k$. If, however, $h_{v^*} = k$, then h_{v^*} is set to l^* if B_{v^*} is empty and to $\pi(w)$ otherwise, where w is the vertex identified in line 11 of the pseudo-code. In the sec-

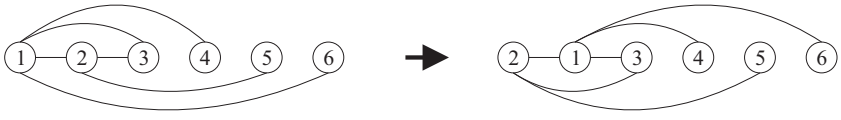

Fig. 8. Shifting the vertex $v = 1$ by one position to the right: the case of $(v, p(l)) = (1, 2) \in E$.

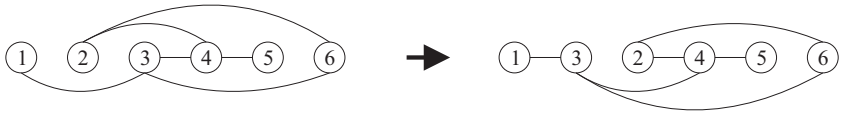

Fig. 9. Shifting the vertex $v = 2$ by one position to the right: the case of $(v, p(l)) = (2, 3) \notin E$.

```
    update_left(v*, k, l*, p)
 1: for i = l* + 1, ..., k do
 2:   u := p(i)
 3:   if (u, v*) ∈ E then
 4:     Update B_u, B_{v*}
 5:     if h_{v*} = i − 1 then Remove v* from L_u end if
 6:     if h_u ⩾ l* then Move u to L_{v*} end if
 7:   end if
 8:   if h_u ⩾ l* then Update h_u end if
 9: end for
10: for i = k + 1, ..., n do
11:   u := p(i)
12:   if l* ⩽ h_u < k and (u, v*) ∈ E then Move u from L_{p(h_u+1)} to L_{v*} end if
13:   if l* ⩽ h_u ⩽ k then Update h_u end if
14: end for
15: if h_{v*} > l* then h_{v*} := l* end if
```

Algorithm 12. Updating L_u, B_u and h_u, $u \in V$, after moving the vertex v^* to the left.

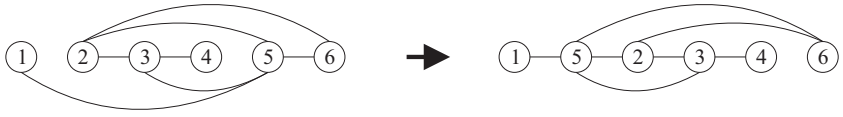

Fig. 10. Relocating the vertex $v^* = 5$ from position 5 to position 2.

```
    update_right(v*, k, l*, p, π)
 1: for i = k, ..., l* − 1, l* + 1, ..., n do
 2:   u := p(i)
 3:   if i < l* and (u, v*) ∈ E then Update B_u, B_{v*} end if
 4:   if h_u ≠ k then
 5:     if k < h_u ⩽ l* then Decrement h_u by 1 end if
 6:   else // the case of h_u = k
 7:     Update h_u and make related modifications to {L_s | s ∈ V}
 8:   end if
 9: end for
10: if h_{v*} = k then
11:   if B_{v*} ≠ ∅ then Add v* to L_w, where w is the leftmost vertex of B_{v*} in p end if
12:   Update h_{v*}
13: end if
```

Algorithm 13. Updating L_u, B_u and h_u, $u \in V$, after moving the vertex v^* to the right.

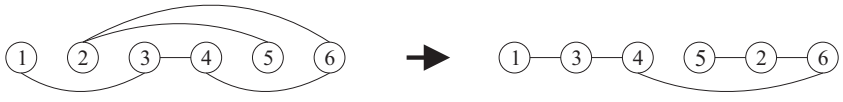

Fig. 11. Relocating the vertex $v^* = 2$ from position 2 to position 5.

ond of these two cases, the vertex v^* is added to L_w. In Figure 11, $B_{v^*} = B_2 = \{5\}$, $w = 5$, $\pi(w) = 4$, $L_5 = \{2\}$ and $h_2 = 4$.

We end the section with the following statement concerning the effectiveness of our local search method.

Proposition 1. *The computational complexity of the* `explore_neighborhood` *procedure is* $O(n^2)$.

Proof. First, consider the `fill_arrays` procedure invoked for each vertex $v \in V$ (lines 2 and 3 in the pseudo-code shown in Algorithm 8). This procedure contains the doubly nested loops in lines 5–14 (see Algorithm 9). We notice, however, that $\Sigma_{v \in V}|L_v| < n$, implying that the inner loop (lines 7–12) is executed less than n times. Since this loop performs $O(n)$ operations it follows that the accumulated complexity of all executions of `fill_arrays` (as well as `clear_arrays`) within `explore_neighborhood` is $O(n^2)$. The rest of the proof is quite obvious. Both `explore_left` and `explore_right` perform $O(n)$ iterations, each of which runs in constant time. Thus, computing the gain Δ^* (lines 2–7 in Algorithm 8) takes $O(n^2)$ operations. From Algorithms 12 and 13, it can be concluded that the running time of each of the procedures `update_left` and `update_right` is $O(n^2)$. Hence, the computational complexity of `explore_neighborhood` is $O(n^2)$ as claimed. □

Table 1
Best solution gaps of MSA-VNS for smaller-sized PMP instances.

Instance	Size	BKV	$\theta = 0$	$\theta = 0.25$	$\theta = 0.5$	$\theta = 0.75$	$\theta = 1$	Best of MSA-VNS
CAN24	24	95[a]	0	0	0	0	0	95
BCSPWR01	39	82[a]	0	0	0	0	0	82
BCSSTK01	48	460[a]	0	0	0	0	0	460
BCSPWR02	49	113[a]	0	0	0	0	0	113
DWT59	59	214[a]	0	0	0	0	0	214
CAN61	61	338[a]	0	0	0	0	0	338
CAN62	62	172[a]	0	0	0	0	0	172
BCSSTK02	66	2145[a]	0	0	0	0	0	2145
DWT66	66	127[a]	0	0	0	0	0	127
DWT72	72	147[a]	0	0	0	0	0	147
CAN73	73	520[a]	0	0	0	0	0	520
ASH85	85	490[a]	0	0	0	0	0	490
DWT87	87	428[a]	0	0	0	0	0	428
CAN96	96	1078[a]	0	0	0	0	0	1078
NOS4	100	651[a]	0	0	0	0	0	651
BCSSTK03	112	272[a]	0	0	0	0	0	272
BCSPWR03	118	434[a]	−11	−11	−11	−11	−11	**423**
BCSSTK04	132	3154[a]	0	0	0	0	0	3154
BCSSTK22	138	628[a]	−10	−10	−10	−10	−10	**618**
CAN144	144	969[a]	0	0	0	0	0	969
BCSSTK05	153	2191[a]	0	0	0	0	1	2191
CAN161	161	2482[a]	−9	−9	−9	−9	−9	**2473**
DWT162	162	1108[a]	2	−1	−1	2	2	**1107**
CAN187	187	2184[a]	−115	−115	−113	−112	−96	**2069**
DWT193	193	4292[b]	−37	−76	−76	−77	−77	**4215**
DWT198	198	1092[a]	−3	−3	−3	−3	−2	**1089**
DWT209	209	2494[a]	−10	−10	−10	−10	−10	**2484**
DWT221	221	1646[a]	−21	−21	−21	−21	−21	**1625**
CAN229	229	3928[a]	−134	−134	−147	−147	−139	**3781**
DWT234	234	782[a]	0	0	0	0	0	782
NOS1	237	467[a]	0	0	0	0	0	467
DWT245	245	2053[a]	−138	−138	−138	−138	−137	**1915**
CAN256	256	5049[a]	−953	−961	−962	−963	−953	**4086**
LSHP265	265	3162[a]	0	0	0	0	0	3162
CAN268	268	5215[a]	−896	−904	−904	−894	−888	**4311**
BCSPWR04	274	1808[a]	32	−33	−33	−33	−27	**1775**
ASH292	292	2717[a]	−44	−44	−44	−44	−43	**2673**
CAN292	292	4673[a]	−743	−763	−763	−763	−749	**3910**
Average			−81.3	−85.1	−85.4	−85.1	−83.4	

[a] Best known values are taken from Table 9 in Sánchez-Oro et al. (2015).
[b] Indicates the best known value reported in Koohestani and Poli (2014).

4. Computational experiments

In this section, we report on the performance of our approach on a set of benchmark PMP instances. The goal of experimentation is to demonstrate that a combination of simulated annealing and variable neighborhood search is an attractive alternative to existing metaheuristic-based algorithms in the current literature on the PMP.

4.1. Experimental setup

The described algorithm has been coded in the C++ programming language, and the tests have been carried out on a PC with an Intel Core 2 Duo CPU running at 3.0 GHz. As a testbed for experimental evaluation of algorithm performance, we considered a set of instances in the Harwell-Boeing Sparse Matrix Collection (Matrix Market, 2011). The set we selected includes matrices of this collection used by Sánchez-Oro et al. (2015) and, in addition, 5 larger matrices used by Koohestani and Poli (2014); 2015). The total number of instances in our main experiment is 77. In the phase of preliminary testing, we also used a training set for determining the parameter setting of the MSA-VNS algorithm (see the next subsection).

In our main experiment, we run MSA-VNS 10 times on each PMP instance in the selected dataset. Maximum CPU time limits for a run were as follows: 5s for $n \leq 100$, 20s for $100 < n \leq 200$, 100s for $200 < n \leq 300$, 300s for $300 < n \leq 500$, 800s for $500 < n \leq 700$, 1800s for $700 < n \leq 2000$, and 3600s for $n > 2000$. The time limits were chosen based on the preliminary trials of the algorithm. The idea was to allow the algorithm to perform a few restarts of SA. For $\theta = 0.5$, the number of SA restarts under the above listed time limits was in the range from 3 to 5 for the majority of the tested problem instances. We will measure the performance of the algorithm in terms of the objective function value of the best solution out of 10 runs as well as the average objective function value of 10 solutions.

4.2. The choice of parameters

The main parameters of SA are the cooling factor α and the number of moves, m, to be attempted at each temperature level. As mentioned in Section 2, these parameters in our experiments were fixed at values recommended in SA literature. Specifically, we set $\alpha = 0.95$ and $m = 100n$. In order to find good parameter settings for the VNS component, we examined the MSA-VNS algorithm's performance on a training sample consisting of 32 symmetric matrices taken from the University of Florida Sparse Matrix Collection (Davis and Hu, 2011). For each parameter setting, we run the MSA-VNS algorithm once for each instance in the training set. We applied the same cutoff times as in the main experiment (see

Table 2
Best solution gaps of MSA-VNS for larger-sized PMP instances.

Instance	Size	BKV	$\theta = 0$	$\theta = 0.25$	$\theta = 0.5$	$\theta = 0.75$	$\theta = 1$	Best of MSA-VNS
DWT307	307	6550[a]	−338	−378	−377	−378	−355	**6172**
DWT310	310	2630[b]	0	0	0	0	0	2630
DWT346	346	6051[b]	−263	−263	−263	−261	−256	**5788**
DWT361	361	4635[b]	−4	−4	−4	−4	−4	**4631**
PLAT362	362	9150[b]	−940	−944	−944	−943	−929	**8206**
LSHP406	406	5964[b]	−9	−9	−9	−9	−9	**5955**
DWT419	419	6619[a]	−544	−546	−546	−544	−522	**6073**
BCSSTK06	420	13239[b]	−278	−395	−410	−399	−386	**12829**
BCSPWR05	443	3076[b]	95	−458	−468	−467	−435	**2608**
CAN445	445	15494[b]	−885	−1295	−1294	−1289	−1273	**14199**
NOS5	468	20446[b]	−542	−550	−550	−547	−516	**19896**
BCSSTK20	485	3006[b]	222	−337	−404	−396	−141	**2602**
DWT492	492	3361[b]	−413	−556	−550	−550	−547	**2805**
494BUS	494	3499[b]	−529	−757	−898	−907	−841	**2592**
DWT503	503	12544[a]	−956	−1116	−1116	−1114	−1087	**11428**
DWT512	512	3975[b]	−124	−155	−152	−152	−99	**3820**
LSHP577	577	10035[b]	0	0	0	0	0	10035
DWT592	592	9046[a]	−230	−230	−230	−230	−230	**8816**
DWT607	607	13278[b]	−731	−815	−847	−845	−783	**12431**
CAN634	634	28493[b]	−2984	−3255	−3296	−3323	−3263	**25170**
662BUS	662	8962[b]	−2202	−2909	−2873	−2968	−2881	**5994**
NOS6	675	9095[b]	0	0	0	0	10	9095
685BUS	685	8528[b]	−1296	−2535	−2513	−2513	−2419	**5993**
CAN715	715	24414[b]	−3823	−4134	−4115	−4122	−4035	**20280**
NOS7	729	34226[b]	−116	−116	−116	−116	−115	**34110**
DWT758	758	6392[b]	−28	−28	−28	−28	−17	**6364**
LSHP778	778	15719[b]	−67	−67	−67	−67	−63	**15652**
BCSSTK19	817	7638[b]	2596	−389	−398	−396	−393	**7240**
DWT869	869	13107[b]	33	−1297	−1301	−1301	−1243	**11806**
DWT878	878	17259[b]	−355	−356	−356	−356	−218	**16903**
GR 30 30	900	24311[b]	−475	−475	−475	−475	−277	**23836**
DWT918	918	16189[a]	−1013	−1182	−1167	−1163	−1069	**15007**
NOS2	957	1907[b]	0	537	617	974	2831	1907
NOS3	960	38676[b]	5663	−2732	−2732	−2673	−2760	**35916**
DWT992	992	31940[c]	10452	−320	−320	−296	−276	**31620**
DWT1005	1005	30488[a]	−857	606	652	658	1220	**29631**
DWT1007	1007	16965[c]	1915	1915	1915	1915	1978	18880
DWT1242	1242	32440[c]	9379	−684	−624	−623	−377	**31756**
DWT2680	2680	84854[a]	101803	168	−2133	−2279	−890	**82575**
Average			2875.8	−668.2	−728.0	−722.7	−581.3	

[a] Indicates the best known value reported in Koohestani and Poli (2014).
[b] Best known values are taken from Table 9 in Sánchez-Oro et al. (2015).
[c] Indicates the best known value reported in Lewis (1994).

Section 4.1). To select the values of the parameters λ, M and $k_{\min}$, we adopted a simple procedure. We allowed one parameter to take a number of predefined values, while keeping the other parameters fixed at reasonable values chosen during preliminary experimentation. We started by investigating the parameter λ, when M was set to 5 and $k_{\min}$ was set to 1. We examined six values of λ: 5,10,25,50,100 and 200. We have found that the algorithm is not very sensitive to changes in λ in the range of 5 to 100. Slightly better solutions were obtained for $\lambda = 50$. The results were significantly worse when the parameter λ was increased to 200. We decided to fix λ at 50. The next step was to assess the influence of the parameter M on the quality of solutions. We compared the results for $M = 5$ with those for the following values of M: 2,3,4,7 and 10. The best performance of MSA-VNS was for $M = 5$. Similar but marginally worse results were achieved in the cases of $M = 4$ and $M = 7$. The choice of other values of M led to a more significant decrease in the performance of the algorithm. On the basis of these results, we fixed M at 5. In our final preliminary experiment, we studied the parameter $k_{\min}$. We investigated the values of $k_{\min}$ in the range 1 to 10. We have found that MSA-VNS is quite robust to variations in this parameter. For the main experiments, we elected to set $k_{\min}$ to 1.

4.3. Numerical results

We conducted computational experiments to evaluate the performance of the five configurations of the MSA-VNS algorithm, specified by the following values of the parameter θ: 0,0.25,0.5,0.75 and 1. As a reference point for comparison between the configurations, we use the best known values reported in the literature. Most of these values are taken from Table 9 in Sánchez-Oro et al. (2015). For the best results for 11 problems in the selected dataset, we refer to Koohestani and Poli (2014) and Lewis (1994). We compute the gap, Γ_{best}, of the value of the best solution out of 10 runs (as well as the gap, Γ_{av}, of the average value of 10 solutions) found by MSA-VNS to the best known value (BKV). Such an assessment method is stronger than comparing the results of an algorithm under investigation against the results of one or more known algorithms individually.

The best solution gaps (Γ_{best}) of the MSA-VNS algorithm are reported in Tables 1 and 2. The first three columns of these tables show the name of the PMP instance, the number of vertices, and the best known value. The next five columns give the gaps Γ_{best} of the tested configurations of MSA-VNS. The last column provides, for each instance, the objective function value of the best solution over all configurations. The values better than BKV are shown in

Table 3
Average solution gaps and time (in seconds) of MSA-VNS for smaller-sized PMP instances.

Instance	$\theta = 0$	$\theta = 0.25$	$\theta = 0.5$	$\theta = 0.75$	$\theta = 1$	Time for $\theta = 0.75$
CAN24	0	0	0	0	0	0.1
BCSPWR01	0	0	0	0	0	2.1
BCSSTK01	0	0	0	0	2.0	3.7
BCSPWR02	0	0	0	0	0	0.7
DWT59	0	0	0	0	0	0.6
CAN61	0	0	0	0	0.2	1.1
CAN62	0	0	0	0	0	0.9
BCSSTK02	0	0	0	0	0	0.0
DWT66	0	0	0	0	0	0.5
DWT72	0	0	0	0	1.4	2.3
CAN73	0	0	0	0	0.4	2.3
ASH85	0	0	0	0	0.9	3.6
DWT87	0	0	0	0	0.6	2.1
CAN96	0	0	0	0	1.0	2.0
NOS4	0	0	0	0	0.1	2.4
BCSSTK03	22.4	0	0	0	0	3.0
BCSPWR03	12.0	−1.4	−4.3	−8.2	−6.8	14.5
BCSSTK04	0	0	0	0.1	1.3	13.2
BCSSTK22	−1.1	−0.5	−4.9	−5.1	5.0	15.8
CAN144	0	0	0	0	0	2.3
BCSSTK05	0.3	0.2	0.3	0.4	1.7	16.9
CAN161	−9.0	−9.0	−9.0	−7.2	−2.2	12.3
DWT162	109.1	48.9	13.7	14.0	2.9	10.7
CAN187	−81.5	−86.8	−85.8	−86.8	−63.8	17.3
DWT193	−8.6	−67.5	−66.8	−75.2	−72.8	14.8
DWT198	−3.0	−2.7	−2.7	−2.8	−1.0	16.2
DWT209	129.3	−9.8	−9.6	−9.6	−8.2	67.3
DWT221	207.9	−20.8	−20.9	−21.0	−20.1	56.6
CAN229	−105.7	−124.6	−131.8	−130.5	−122.3	86.8
DWT234	0	0	0	1.5	1.6	77.2
NOS1	0	0	0	0	121.0	72.6
DWT245	93.1	−133.2	−134.9	−135.3	−130.6	76.0
CAN256	−446.9	−956.9	−956.6	−956.1	−946.3	93.2
LSHP265	0	0	0	0	0	30.1
CAN268	−673.7	−849.0	−852.8	−877.4	−872.7	85.4
BCSPWR04	444.5	95.5	16.1	2.3	0.2	85.5
ASH292	389.0	−32.6	−38.5	−38.6	−31.3	83.7
CAN292	−327.1	−537.5	−574.1	−660.0	−641.3	85.3
Average	−6.6	−70.7	−75.3	−78.8	−73.1	

boldface. The results, averaged over all problem instances, are presented in the last row.

Table 1 summarizes the results for $n \leq 300$. We can see that the combination of MSA and VNS outperforms each algorithm taken separately. We also observe that all configurations of MSA-VNS were able to improve BKV for many instances in the dataset. The configurations with $\theta = 0.25$ and $\theta = 0.5$ have been more successful in this respect. Each of them improved BKV for 16 problem instances. The new BKVs are highlighted in bold font in the last column of the table.

The results for $n > 300$ are shown in Table 2. From the last line, we see that the variant of MSA-VNS with $\theta = 0.5$ is the best and that with $\theta = 0.75$ is the second best performing approach. The MSA algorithm alone (represented by the column under heading $\theta = 1$) is inferior to each tested configuration of MSA-VNS with $\theta \in (0, 1)$. We also find that VNS (represented by the fourth column) is the worst algorithm in the comparison. It should be noticed, however, that the positive average of Γ_{best} for VNS in the last line of the table is due to poor performance of this algorithm on the three large instances, DWT992, DWT1242 and DWT2680. Observation of the last column indicates that, using MSA-VNS, we improved BKV for 34 instances out of 39.

In Tables 3 and 4, we report the average solution gaps (Γ_{av}) of our algorithm. In addition, in the last column of each table, we display the average running time to the best solution in a run for the configuration of MSA-VNS with $\theta = 0.75$. The running times taken by other configurations of the algorithm were very similar and, to save space, are not provided. On the basis of the results obtained, we find that MSA-VNS with $\theta = 0.75$ is superior to the other four variants of the algorithm. Among them, MSA-VNS with $\theta = 0.5$ is ranked second. We note that also MSA performed quite well achieving negative average value of Γ_{av}. The worst results were obtained from using our VNS implementation. Another observation from Tables 3 and 4 is that, for the most successful MSA-VNS variants, the average solution values in many cases are better than the BKVs reported in the literature. For example, the number of such cases for MSA-VNS with $\theta = 0.75$ is 44 (equals the number of negative entries in the columns for this MSA-VNS configuration).

For indicative purpose, we compare the results produced by our MSA-VNS approach with those obtained by the following state-of-the-art algorithms in the literature: S*+, the hybrid of local search and the best algorithm created by the genetic hyper-heuristic from Koohestani and Poli (2014); GP, the genetic programming system from Koohestani and Poli (2015); SS, the scatter search approach from Sánchez-Oro et al. (2015). The results are summarized in Tables 5 (for $n \leq 300$) and 6 (for $n > 300$). Like in the previous tables, the performance measure adopted is the solution gap to the best known value. Columns 3–5 are prepared using data taken from the respective sources. Our approach is represented by the MSA-VNS configuration with $\theta = 0.75$.

Inspection of Tables 5 and 6 reveals that the solutions found by MSA-VNS are better than those produced by other algorithms included in the comparison. We draw this conclusion by comparing, for each instance, the objective function values of the best solutions found by MSA-VNS, S*+ and GP, and the average objective

Table 4
Average solution gaps and time (in seconds) of MSA-VNS for larger-sized PMP instances.

Instance	$\theta = 0$	$\theta = 0.25$	$\theta = 0.5$	$\theta = 0.75$	$\theta = 1$	Time for $\theta = 0.75$
DWT307	−256.2	−341.1	−354.2	−359.7	−328.8	266.2
DWT310	0	0	0	0	0	42.5
DWT346	−232.0	−259.5	−258.8	−258.5	−248.6	272.5
DWT361	−4.0	−4.0	−4.0	−4.0	−3.6	181.3
PLAT362	−331.9	−940.3	−939.6	−936.5	−919.8	276.8
LSHP406	−9.0	−9.0	−9.0	−9.0	−8.9	114.7
DWT419	844.9	−529.2	−530.0	−532.3	−497.6	270.6
BCSSTK06	1956.3	−367.5	−376.9	−366.9	−351.8	272.4
BCSPWR05	703.9	−365.9	−394.1	−398.8	−389.1	276.5
CAN445	−391.7	−1216.3	−1238.4	−1249.6	−1208.5	275.7
NOS5	513.3	−520.9	−522.7	−525.0	−486.6	268.3
BCSSTK20	898.6	−41.6	−201.8	−170.4	4.9	291.1
DWT492	1773.4	−414.2	−532.1	−523.3	−503.9	269.1
494BUS	226.8	−591.0	−777.1	−799.6	−759.0	287.5
DWT503	405.2	−1088.3	−1110.5	−1107.6	−1068.5	753.7
DWT512	403.5	−93.3	−108.7	−117.6	−73.9	696.0
LSHP577	697.7	0	0	0	1.3	457.8
DWT592	941.4	−223.8	−229.4	−229.0	−223.5	576.7
DWT607	836.6	−780.2	−783.2	−781.4	−748.7	739.9
CAN634	−640.1	−3168.2	−3201.3	−3204.5	−3127.9	767.1
662BUS	−842.3	−2623.0	−2681.7	−2736.9	−2703.9	762.4
NOS6	793.1	0	0	0	14.2	600.2
685BUS	315.2	−2098.1	−2234.8	−2295.4	−2232.8	769.9
CAN715	−487.4	−3928.4	−4062.1	−4058.1	−3953.2	1697.1
NOS7	927.7	−113.9	−116.0	−113.2	−95.9	1380.7
DWT758	3816.2	2074.7	1620.5	1053.0	671.5	1528.1
LSHP778	811.2	−67.0	−67.0	−67.0	−58.4	1364.3
BCSSTK19	7076.0	528.4	269.6	−109.4	−5.0	1651.8
DWT869	4619.6	−966.2	−1137.1	−1187.3	−1088.4	1609.0
DWT878	1839.9	−199.2	−175.7	−189.2	−101.7	1606.3
GR 30 30	−452.4	−431.5	−414.8	−413.5	−153.4	1616.5
DWT918	4894.3	−1132.7	−1117.9	−1110.3	−1029.6	1734.3
NOS2	945.0	1149.0	1099.4	1367.7	3899.4	1783.7
NOS3	19822.3	−2533.0	−2552.1	−2584.0	−2560.8	1739.4
DWT992	32249.2	−171.6	−224.8	−218.0	−208.8	1678.4
DWT1005	4017.3	871.8	940.8	869.7	1462.8	1694.4
DWT1007	6290.0	1915.0	1917.1	1917.2	2041.6	1613.5
DWT1242	18398.4	4684.5	2748.8	1491.6	1224.5	1727.9
DWT2680	140725.5	35940.6	13590.5	13748.3	15678.5	3221.4
Average	6515.3	562.7	−106.9	−159.2	− 3.6	

Table 5
Comparison of the MSA-VNS approach against the S*+, GP and SS algorithms on the smaller-sized PMP instances: solution gaps to BKV (average solution gaps of MSA-VNS are shown in parentheses).

Instance	BKV	S*+ (Koohestani and Poli, 2014)[a]	GP (Koohestani and Poli, 2015)[a]	SS (Sánchez-Oro et al., 2015)[b]	MSA-VNS ($\theta = 0.75$)
CAN24	95	n/a	n/a	0	0(0)
BCSPWR01	82	1	n/a	0	0(0)
BCSSTK01	460	n/a	n/a	6	0(0)
BCSPWR02	113	12	n/a	0	0(0)
DWT59	214	0	5	9	0(0)
CAN61	338	n/a	n/a	0	0(0)
CAN62	172	n/a	n/a	0	0(0)
BCSSTK02	2145	n/a	n/a	0	0(0)
DWT66	127	0	0	0	0(0)
DWT72	147	17	14	4	0(0)
CAN73	520	n/a	n/a	0	0(0)
ASH85	490	n/a	n/a	0	0(0)
DWT87	428	33	5	6	0(0)
CAN96	1078	n/a	n/a	2	0(0)
NOS4	651	n/a	n/a	0	0(0)
BCSSTK03	272	n/a	n/a	0	0(0)
BCSPWR03	434	9	n/a	0	−11(-8.2)
BCSSTK04	3154	n/a	n/a	5	0(0.1)
BCSSTK22	628	n/a	n/a	13	−10(-5.1)
CAN144	969	n/a	n/a	0	0(0)
BCSSTK05	2191	n/a	n/a	1	0(0.4)
CAN161	2482	n/a	n/a	0	−9(-7.2)
DWT162	1108	236	236	178	2(14.0)
CAN187	2184	n/a	n/a	11	−112(-86.8)
DWT193	4292	0	119	96	−77(-75.2)
DWT198	1092	n/a	n/a	0	−3(-2.8)
DWT209	2494	515	422	127	−10(-9.6)
DWT221	1646	90	109	0	−21(-21.0)
CAN229	3928	n/a	n/a	213	−147(-130.5)
DWT234	782	n/a	n/a	21	0(1.5)
NOS1	467	n/a	n/a	0	0(0)
DWT245	2053	757	328	0	−138(-135.3)
CAN256	5049	n/a	n/a	0	−963(-956.1)
LSHP265	3162	n/a	n/a	0	0(0)
CAN268	5215	n/a	n/a	0	−894(-877.4)
BCSPWR04	1808	667	n/a	184	−33(2.3)
ASH292	2717	n/a	n/a	67	−44(-38.6)
CAN292	4673	n/a	n/a	45	−763(-660.0)
Average		179.8	137.6	26.0	−85.1(-78.8)

n/a — Data not available.
[a] Best results in 20 runs.
[b] Each entry corresponds to a single run of SS.

function value reported by MSA-VNS (see solution gaps in parentheses in the last column) with the objective function value of the solution obtained in a single run of SS. As we can see from the last row of each of the tables, the average solution gap to BKV is negative for MSA-VNS and positive for other algorithms. Additionally, we performed statistical analysis of experimental results. We used the Wilcoxon signed-rank test to determine if there are significant differences between algorithms in the quality of the solutions obtained. We applied this test to the pairs consisting of MSA-VNS and one of the other three algorithms. Comparing the results in column 5 with those in parentheses in the last column of Tables 5 and 6, we find that SS yielded better solutions than MSA-VNS for two instances in the dataset, DWT758 and NOS2. The sum of ranks in favor of SS is 93, while that in favor of MSA-VNS is 1503. The Wilcoxon signed-rank test shows that the difference between results of MSA-VNS and SS is significant with p-value smaller than 0.001. Both S*+ and GP produced a better solution than MSA-VNS for only one instance, namely DWT1005. The rank assigned to this instance is 16 in the case of the pair S*+ versus MSA-VNS and 13 in the case of the pair GP versus MSA-VNS. The sum of ranks for MSA-VNS in these two cases is 362 and 263, respectively. By applying the Wilcoxon signed-rank test, we find that differences between MSA-VNS and both S*+ and GP are statistically significant (p-value is less than 0.001).

We remark, however, that it is rather difficult to compare experimental results of MSA-VNS against those obtained by the algorithms proposed in Koohestani and Poli (2014, 2015); Sánchez-Oro et al. (2015), mostly because the runs were performed on different computers. Therefore, naturally, the results of comparison should be interpreted with care. It is worth noting, however, that there are no drastic differences in the execution times between MSA-VNS, S*+, GP and SS. For example, for the DWT245 instance, MSA-VNS took 76s (Table 3), GP took roughly 69s on an AMD Athlon (tm) dual-core processor 2.20 GHz (Koohestani and Poli, 2015), SS took 29.7s on an Intel Core i7 2600 machine running at 3.4 GHz (Sánchez-Oro et al., 2015), and S*+ was allowed to run for 1000s (like for all other tested problem instances) on the same computer as the one used for GP (Koohestani and Poli, 2014). For the largest instance considered in all studies, DWT918, the execution times were 1734.3s, 610.1s, 1277.3s and 1000s for MSA-VNS, GP, SS and S*+, respectively.

Tables 5 and 6 show that MSA-VNS is a good option for addressing the PMP in situations where computation time is not a critical issue. The statistical analysis supports this statement. To reduce the computational cost of direct and iterative methods to

Table 6
Comparison of the MSA-VNS approach against the S*+, GP and SS algorithms on the larger-sized PMP instances: solution gaps to BKV (average solution gaps of MSA-VNS are shown in parentheses).

Instance	BKV	S*+(Koohestani and Poli, 2014)[a]	GP(Koohestani and Poli, 2015)[a]	SS(Sánchez-Oro et al., 2015)[b]	MSA-VNS($\theta = 0.75$)
DWT307	6550	0	418	126	−378(-359.7)
DWT310	2630	10	36	0	0(0)
DWT346	6051	n/a	n/a	0	−261(-258.5)
DWT361	4635	64	64	0	−4(-4.0)
PLAT362	9150	n/a	n/a	1470	−943(-936.5)
LSHP406	5964	n/a	n/a	0	−9(-9.0)
DWT419	6619	0	539	60	−544(-532.3)
BCSSTK06	13239	n/a	n/a	198	−399(-366.9)
BCSPWR05	3076	208	n/a	278	−467(-398.8)
CAN445	15494	n/a	n/a	0	−1289(-1249.6)
NOS5	20446	n/a	n/a	0	−547(-525.0)
BCSSTK20	3006	n/a	n/a	0	−396(-170.4)
DWT492	3361	n/a	n/a	0	−550(-523.3)
494BUS	3499	n/a	n/a	0	−907(-799.6)
DWT503	12544	0	534	608	−1114(-1107.6)
DWT512	3975	n/a	n/a	0	−152(-117.6)
LSHP577	10035	n/a	n/a	10	0(0)
DWT592	9046	0	555	452	−230(-229.0)
DWT607	13278	n/a	n/a	0	−845(-781.4)
CAN634	28493	n/a	n/a	0	−3323(-3204.5)
662BUS	8962	n/a	n/a	0	−2968(-2736.9)
NOS6	9095	n/a	n/a	0	0(0)
685BUS	8528	n/a	n/a	0	−2513(-2295.4)
CAN715	24414	n/a	n/a	0	−4122(-4058.1)
NOS7	34226	n/a	n/a	449	−116(-113.2)
DWT758	6392	100	219	0	−28(1053.0)
LSHP778	15719	n/a	n/a	0	−67(-67.0)
BCSSTK19	7638	n/a	n/a	0	−396(-109.4)
DWT869	13107	55	284	0	−1301(-1187.3)
DWT878	17259	1448	1057	0	−356(-189.2)
GR 30 30	24311	n/a	n/a	0	−475(-413.5)
DWT918	16189	0	778	313	−1163(-1110.3)
NOS2	1907	n/a	n/a	0	974(1367.7)
NOS3	38676	n/a	n/a	6955	−2673(-2584.0)
DWT992	31940	516	202	n/a	−296(-218.0)
DWT1005	30488	0	1	n/a	658(869.7)
DWT1007	16965	3500	2678	n/a	1915(1917.2)
DWT1242	32440	1661	7538	n/a	−623(1491.6)
DWT2680	84854	0	3549	n/a	−2279(13748.3)
Average		472.6	1230.1	321.1	−722.7(-159.2)

n/a — Data not available.
[a] Best results in 20 runs.
[b] Each entry corresponds to a single run of SS.

solve sparse linear systems, typically very fast profile minimization algorithms are required. In such cases, a suitable strategy is to apply fast heuristic techniques for the PMP, for example, an enhanced version of the Sloan algorithm, which was produced using a genetic hyper-heuristic approach (see Koohestani and Poli, 2014). However, as it was already alluded to in the introduction, there are scenarios in solving sparse linear systems where metaheuristic-based profile minimization algorithms are useful. One such scenario is to run an algorithm for the PMP independently of direct sparse matrix methods. This is appropriate when the same coefficient matrix is used repeatedly or there are a large number of coefficient matrices whose sparsity pattern is represented by the same graph. In such situations, our MSA-VNS algorithm is expected to be quite practical. Furthermore, there are applications (for example, Meijer and de Pol, 2015; Mueller et al., 2007) requiring the solution of the PMP where no hard limit on the run time of the algorithms is imposed. The MSA-VNS algorithm may be a quite reasonable choice when dealing with such applications.

In closing this section, we should mention that the C++ code implementing the MSA-VNS algorithm for the PMP is made publicly available as a benchmark for future comparisons, see Palubeckis (2016).

5. Concluding remarks

In this paper, we have developed a method for the PMP, which combines a variable neighborhood search scheme with the simulated annealing technique. We have implemented the latter as a multi-start algorithm, in which starting solutions are generated randomly. The solution delivered by this algorithm is submitted as input to the VNS component of the method. Our VNS implementation uses a local search algorithm which relies on a fast insertion neighborhood exploration procedure. We have shown that the time complexity of this procedure is $O(n^2)$.

We have conducted computational experiments for our algorithm on a set of well-recognized benchmark PMP instances in the literature. One of the tasks examined during experimentation was appropriately dividing the time limit for MSA-VNS execution into two intervals, one for MSA and another for VNS. We have found that it is advantageous to give between 50 and 75% of the time to MSA and the rest to VNS. Based on the results of experiments, we can conclude that the MSA-VNS algorithm is superior to the current state-of-the-art metaheuristic approaches for the PMP. Using MSA-VNS, we improved the best known solutions for 50 benchmark PMP instances out of 77.

We believe that efforts should be continued to develop fast algorithms for the PMP that would be capable of producing so-

lutions of excellent quality. The described local search procedure could be a useful ingredient worth embedding in such algorithms. Our empirical results show that the PMP is a difficult optimization problem. Therefore, we think that it can serve as a benchmark problem for evaluating general-purpose metaheuristic approaches in the combinatorial optimization area.

References

Armstrong, B.A., 1985. Near-minimal matrix profiles and wavefronts for testing nodal resequencing algorithms.. Int J Numer Methods Eng 21 (10), 1785–1790.

Barnard, S., Pothen, A., Simon, H., 1995. A spectral algorithm for envelope reduction of sparse matrices.. Numer Linear Algebra Appl 2 (4), 317–334.

Bernardes J. .A, B., de Oliveira SL, G., 2015. A systematic review of heuristics for profile reduction of symmetric matrices.. Procedia Comput Sci 51, 221–230.

Berry M., W., Hendrickson, B., Raghavan, P., 1996. Sparse matrix reordering schemes for browsing hypertext.. In: Renegar J, Shub M, Smale S, editors. Lectures in Applied Mathematics. Vol 32: The Mathematics of Numerical Analysis.. American Mathematical Society Press, Park City, Utah, USA, pp. 99–123.

Bolanos M., E., Aviyente, S., Radha, H., 2012. Graph entropy rate minimization and the compressibility of undirected binary graphs.. In: Proceedings of IEEE Statistical Signal Processing Workshop (SSP). Ann Arbor, MI, USA, pp. 109–112.

Campos, V., Piñana, E., Martí, R., 2011. Adaptive memory programming for matrix bandwidth minimization.. Ann Oper Res 183, 7–23.

Černý, V., 1985. Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm.. J Optim Theory Appl 45 (1), 41–51.

Cuthill, E., McKee, J., 1969. Reducing the bandwidth of sparse symmetric matrices.. In: Proceedings of the ACM 24th National Conference, pp. 157–172.

Davis, T.A., Hu, Y., 2011. The university of florida sparse matrix collection.. ACM Trans Math Softw 38 (1), 1–25.

Davis, T.A., Rajamanickam, S., Sid-Lakhdar, W.M., 2016. A survey of direct methods for sparse linear systems.. Acta Numer 25, 383–566.

Duff, I.S., Reid, J.K., Scott, J.A., 1989. The use of profile reduction algorithms with a frontal code.. Int J Numer Methods Eng 28 (11), 2555–2568.

George, J.A., 1971. Computer implementation of the finite element method. Computer Science Department, Stanford University, Stanford, CA, USA ph.d. thesis.

Gibbs, N.E., 1976. Algorithm 509: A hybrid profile reduction algorithm.. ACM Trans Math Softw 2 (4), 378–387.

Gibbs, N.E., Poole, W.G., Stockmeyer, P.K., 1976. An algorithm for reducing the bandwidth and profile of a sparse matrix.. SIAM J Numer Anal 13 (2), 236–250.

Guan, Y., Williams, K.L., 2003. Profile minimization on triangulated triangles.. Discrete Math 260 (1–3), 69–76.

Hager, W.W., 2002. Minimizing the profile of a symmetric matrix.. SIAM J Sci Comput 23 (5), 1799–1816.

Hansen, P., Mladenović, N., 2001. Variable neighborhood search: principles and applications.. Eur J Oper Res 130 (3), 449–467.

Hansen, P., Mladenović, N., Moreno Pérez, J.A., 2010. Variable neighbourhood search: methods and applications.. Ann Oper Res 175, 367–407.

Higham, D.J., 2003. Unravelling small world networks.. J Comput Appl Math 158 (1), 61–74.

Hu, Y.F., Scott, J.A., 2001. A multilevel algorithm for wavefront reduction.. SIAM J Sci Comput 23 (4), 1352–1375.

Isazadeh, A., Izadkhah, H., Mokarram, A.H., 2012. A learning based evolutionary approach for minimization of matrix bandwidth problem.. Appl Math Inform Sci 6 (1), 51–57.

Kaveh, A., Sharafi, P., 2012. Ordering for bandwidth and profile minimization problems via charged system search algorithm.. IJST–T Civ Eng 36 (C1), 39–52.

Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P., 1983. Optimization by simulated annealing. Science 220 (4598), 671–680.

Koohestani, B., Poli, R., 2014. Evolving an improved algorithm for envelope reduction using a hyper-heuristic approach.. IEEE Trans Evol Comput 18 (4), 543–558.

Koohestani, B., Poli, R., 2015. Addressing the envelope reduction of sparse matrices using a genetic programming system.. Comput Optim Appl 60 (3), 789–814.

Kumfert, G., Pothen, A., 1997. Two improved algorithms for envelope and wavefront reduction.. BIT 37 (3), 559–590.

Kuo, D., Chang, G.J., 1994. The profile minimization problem in trees.. SIAM J Comput 23 (1), 71–81.

Lewis, R.R., 1994. Simulated annealing for profile and fill reduction of sparse matrices.. Int J Numer Methods Eng 37 (6), 905–925.

Lin, Y., Yuan, J., 1994. Profile minimization problem for matrices and graphs.. Acta Math Appl Sin–E 10 (1), 107–112.

Mafteiu-Scai LO, 2014. The bandwidths of a matrix. a survey of algorithms.. An Univ Vest Timis Ser Mat–Inform 52 (2), 183–223.

Matrix Market, 2011. http://math.nist.gov/MatrixMarket/collections/hb.html [accessed 19.12.16].

Meijer, J., de Pol, J., 2015. Bandwidth and wavefront reduction for static variable ordering in symbolic model checking.ArXiv preprint. arXiv:1511.08678v1 [cs.SE].

Mladenovic, N., Urosevic, D., Pérez-Brito, D., García-González, C.G., 2010. Variable neighborhood search for bandwidth reduction.. Eur J Oper Res 200 (1), 14–27.

Mueller, C., Martin, B., Lumsdaine, A., 2007. A comparison of vertex ordering algorithms for large graph visualization.. In: Proceedings of the 6th International Asia-Pacific Symposium on Visualization (APVIS'07). Sydney, Australia, pp. 141–148.

Palubeckis, G., 2015. Fast local search for single row facility layout.. Eur J Oper Res 246 (3), 800–814.

Palubeckis, G., 2016. The profile minimization problem. http://www.personalas.ktu.lt/~ginpalu/pmp.html[accessed 10.01.17].

Palubeckis, G., 2017. Single row facility layout using multi-start simulated annealing.. Comput Ind Eng 103, 1–16.

Paulino, G.H., Menezes, I.F.M., Gattass, M., Mukherjee, S., 1994. Node and element resequencing using the Laplacian of a finite element graph: Part I– general concepts and algorithm.. Int J Numer Methods Eng 37 (9), 1511–1530.

Paulino, G.H., Menezes, I.F.M., Gattass, M., Mukherjee, S., 1994. Node and element resequencing using the Laplacian of a finite element graph: Part II – implementation and numerical results.. Int J Numer Methods Eng 37 (9), 1531–1555.

Pop, P., Matei, O., Comes, C.A., 2014. Reducing the bandwidth of a sparse matrix with a genetic algorithm.. Optimization 63 (12), 1851–1876.

Reid, J.K., Scott, J.A., 1999. Ordering symmetric sparse matrices for small profile and wavefront.. Int J Numer Methods Eng 45 (12), 1737–1755.

Reid, J.K., Scott, J.A., 2002. Implementing Hager's exchange methods for matrix profile reduction.. ACM Trans Math Softw 28 (4), 377–391.

Rutenbar, R.A., 1989. Simulated annealing algorithms: an overview.. IEEE Circuits Devices Mag 5 (1), 19–26.

Sánchez-Oro, J., Laguna, M., Duarte, A., Martí, R., 2015. Scatter search for the profile minimization problem.. Networks 65 (1), 10–21.

Sloan, S.W., 1986. An algorithm for profile and wavefront reduction of sparse matrices.. Int J Numer Methods Eng 23 (2), 239–251.

Tewarson, R.P., 1973. Sparse matrices.. Academic Press;, New York.

Torres-Jimenez, J., Izquierdo-Marquez, I., Garcia-Robledo, A., Gonzalez-Gomez, A., Bernal, J., Kacker, R.N., 2015. A dual representation simulated annealing algorithm for the bandwidth minimization problem on graphs.. Inf Sci 303, 33–49.

Xu, S., Xue, W., Lin, H.X., 2013. Performance modeling and optimization of sparse matrix-vector multiplication on NVIDIA CUDA platform.. J Supercomput 63 (3), 710–721.