Applied Soft Computing 36 (2015) 185–203

Contents lists available at ScienceDirect

Applied Soft Computing

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

A great deluge and tabu search hybrid with two-stage memory support for quadratic assignment problem

Adnan Acan*, Ahmet Ünveren

Eastern Mediterranean University, Computer Engineering Department, Gazimagusa, Turkey

ARTICLE INFO

Article history:
Received 30 October 2014
Received in revised form 18 May 2015
Accepted 26 June 2015
Available online 29 July 2015

Keywords:
Combinatorial optimization
Great deluge algorithm
Memory-based search
Metaheuristics
Quadratic assignment problem
Tabu search

ABSTRACT

A two-stage memory architecture is maintained within the framework of great deluge algorithm for the solution of single-objective quadratic assignment problem. Search operators exploiting the accumulated experience in memory are also implemented to direct the search towards more promising regions of the solution space. The level-based acceptance criterion of the great deluge algorithm is applied for each best solution extracted in a particular iteration. The use of short- and long-term memory-based search supported by effective move operators resulted in a powerful combinatorial optimization algorithm. A successful variant of tabu search is employed as the local search method that is only applied over a few randomly selected memory elements when the second stage memory is updated. The success of the presented approach is illustrated using sets of well-known benchmark problems and evaluated in comparison to well-known combinatorial optimization algorithms. Experimental evaluations clearly demonstrate that the presented approach is a competitive and powerful alternative for solving quadratic assignment problems.

1. Introduction

Combinatorial optimization problems are representative models of many real life optimization tasks. Location of facilities, finding shortest routes, staff and task scheduling, packing and covering and segmentation of images are a few within a large set of combinatorial optimization problems for which development of dedicated solution methods is still an area of hot research. A common nature of many combinatorial optimization problems is their NP-complete type of complexity which directly indicates the computational difficulty of obtaining exact solutions for these problems. In fact, exact solutions are possible for (very) small-size problem instances and finding an approximate solution with close-to-optimum quality is the fundamental problem to be solved in most of the time. Since advanced computational resources enable us to deal with larger-size and more complex problems, development of more efficient solution methodologies, in terms of solution quality and convergence speed, became the ultimate objectives of almost all optimization tasks.

Considering the exploration of a solution space using a particular metaheuristic method employing a set of search operators, the two challenging problems to deal with are early stagnation at a locally optimal solution and efficient guidance towards globally optimal solutions. In this respect, searching around multiple promising solutions, particularly at initial iterations of an algorithm, should clearly be a major concern for the design of search algorithms. For this purpose, memory methods using an archive of a number of promising solutions, or reference points, are proposed and they are shown to be useful through experimental evaluations. Accordingly, a memory-based search procedure has two fundamental objectives; one is to intensify the search around the potentially promising solutions found so far while the second is diversifying the exploration within the solution space by using more than one reference template within the maintained memory. This way, early stagnation at a locally optimal solution can be avoided by providing alternative directions for search and more efficient guidance towards globally optimal solutions can be provided by appropriate memory-based search operators and memory update methods.

Great deluge algorithm (GDA), which is first proposed by Dueck [1], is a local search algorithm similar to simulated annealing (SA) with the exception that acceptance rule is deterministic and controlled by a threshold parameter called the *Level*. It uses two other algorithmic parameters, namely, the number of iterations and an estimation of the best fitness value of the objective function. In its

* Corresponding author. Tel.: +90 392 6301068; fax: +90 392 3650711.
E-mail addresses: adnan.acan@emu.edu.tr (A. Acan), ahmet.unveren@emu.edu.tr (A. Ünveren).

http://dx.doi.org/10.1016/j.asoc.2015.06.061
1568-4946/

search in a solution space, GDA accepts a new solution if its fitness is better than the best solution found so far or the fitness is less than a dynamically updated upper bound, which is the *Level*. Estimated best fitness value of the objective function is usually set less than or equal to the fitness of the best solution known so far. *Level* is initially set to the fitness of the initial solution and is lowered iteratively by an additive parameter β computed as a function the initial value of *Level* and the estimated best fitness value of the objective function.

After its initial introduction, improved versions of GDA are proposed in several of publications. These improvements can be considered in two categories: changes in basic algorithm parameters and search operators, and hybridizations with other metaheuristics. Since GDA has basically two parameters, namely the *Level* and the level decay rate β, algorithmic improvements based on different ways of changing these parameters are the main concern of studies in the first category. Burke et al. implemented a linear decreasing of decay rate *beta* such that the amount of decay is computed based on the maximum number of iterations. The authors called this strategy as degraded ceiling algorithm [2]. According to published results, this strategy produced most of the best fitness values on examination timetabling benchmark problems. Later, Burke and Bykov developed the flex-deluge algorithm (FGDA) in which the acceptance of uphill moves is controlled by a flexibility parameter [3]. For the solution of exam timetabling problems, FGDA was claimed to provide good results particularly for large-scale problems. The modified great deluge algorithm (MGDA), proposed by Ravi [4], introduced a new neighbourhood search and a new level decay mechanism that depends on the amount of improvement achieved when the current solution is modified to get the new one. Basically, level is decreased if fitness of the new solution is below the current level, whereas it is increased otherwise. This approach is applied for reliability optimization and optimal redundancy allocation problems and was observed to perform as well as ant colony optimization, while it was significantly better than simulated annealing algorithm. Silva and Obit developed the nonlinear great deluge algorithm (NLGDA) that uses a nonlinear decay rate for the level parameter [5]. An exponential expression including four parameters is used to determine the functional shape of decay rate. When used for the solution of course timetabling problems, NLGDA updated the best solutions of four benchmark instances among the eleven in the experimental set. The extended great deluge algorithm (EGDA), developed by McMullan, uses the concept of reheating from simulated annealing such that, if no improvement is obtained within a predefined period, the level parameter is reset to the current objective function value [6]. For a set of course timetabling benchmark problems, EGDA produced ten best solutions for five medium-size problems and for one large-size problem. Nahas et al. used EGDA in two steps as follows: EGDA is used for N_1 number of iterations to find a locally optimal solution. Then, best found solution is used as the initial solution for EGDA and improved through another N_2, $N_2 < N_1$, number of iterations [7]. The authors called this approach as iterated great deluge algorithm (IGDA) and, for a set of 48 facility layout problems, IGDA is reported to update the best found results for 17 of them. Ozcan et al. combined reinforcement learning (RL) and the GDA within a hyperheuristic (HH) framework such that RL is used to select one of a set of low-level heuristics, that is applied as a move to generate a new solution, whereas the GDA's level-based decision mechanism is used to accept or reject the move [8]. For a set of thirteen university examination timetabling problems, this approach is observed to be better than simulated annealing and simple random hyper-heuristics for six problem instances.

The second category of the improvements includes hybridizations with other metaheuristics or slight modifications of level-based acceptance mechanism. Milli [9] used GDA as a local search procedure within a genetic algorithm (GA) framework for the solution of course timetabling problems. After each generation, GDA is employed to improve the best solution found so far. The author claimed that the proposed combination generated consistently good results for benchmark problems used in experimental studies. Landa-Silva et al. proposed another hybrid of evolutionary algorithms and NLGDA where an individual selected from the current population by tournament selection is modified by mutation and the mutated individual is improved by NLGDA [10]. If the resulting solution is better than the worst solution in GA's population, it replaces that worst solution. The proposed hybrid approach is claimed to perform better than its competitors in most of the benchmark course timetabling problems used for comparative evaluations. A third GDA-related hybrid approach on the solution of course timetabling problem combines particle collision algorithm (PCA) and GDA such that GDA's level-based acceptance criterion is used in the scattering phase of PCA [11]. The experimental results presented in the paper show that the proposed hybrid approach performs better than its eleven competitors for the solution of eleven widely used course timetabling problem instances. In a sequential implementation of Tabu search and GDA, proposed by Abdullah et al. [12], the current solution is first modified by GDA and then the modified solution is improved by tabu search. Finally, the best of the solutions found by the two algorithms is used to start the next iteration. When used for the solution of course time tabling problems, the proposed method is reported to provide the best solutions for most of the benchmark problems. In another publication, Abdullah et al. combined GDA with an electromagnetic-like mechanism to solve the examination timetabling problem [13]. This population based method is implemented in two phases. In the first phase, a positive amount of charge is assigned to each timetable in the population based on their fitness relative to the fitness of the best timetable found so far. Consequently, the total force on each timetable is computed using an analogy to interaction of charged particles and these total force values are used to calculate the estimated qualities of individual timetables in the population. In the second phase, the level decay parameter of GDA is determined based on the estimated qualities of individual timetables and GDA is applied on each timetable of the population for a number of iterations. Among the eleven examination timetabling instances and against the eight competitors in experimental evaluations, the authors reported that this hybrid approach provided the best known solutions for nine problem instances.

Among a few GDA hybrids used for real-valued function optimization, Ghatei et al. hybridized GDA and particle swarm optimization (PSO) in which GDA is used as final local search procedure to further improve the best solution found by the PSO algorithm [14]. This approach is tested over four benchmark functions only and it is claimed to be better than GAs and PSO.

Considering the above literature review and based on our detailed search for different solution approaches to QAP, a direct application of GDA for the solution of QAP could not be found so far. However, a problem that is most similar to QAP and solved using GDA is the dynamic facility layout problem [7]. In addition to this, even though GDA is not used for the solution of QAP, its success is demonstrated on another subclass of large-size combinatorial optimization problems, namely the timetabling and scheduling problems. Hence, our main reasoning behind the use of GDA for the solution of QAP can be stated as its exhibited success for difficult classes of combinatorial optimization problems, implementation simplicity, adaptive and deterministic decision character, and the objective of demonstrating the effectiveness of a novel two-stage memory architecture within the framework of a metaheuristic for the solution of a provably difficult combinatorial optimization problem. The proposed two-stage memory-based and operator-adapted variation of GDA for QAP, its objectives and comparative level of success are illustrated in the following sections.

The fundamental objective of memory based implementation of a metaheuristic algorithm is to use experience based knowledge stored in memory to localize the globally optimal solutions better while the exploration capability of the underlying search methodology is not weakened. Memory based search strategies have already been implemented within several metaheuristic algorithms and they are shown to be more successful compared to memoryless implementations. In fact, memory-based implementations have first been implemented within the framework of genetic algorithms (GAs). Basically, from GAs point of view, memory based implementations aim to adapt the GAs behavior when either the solution quality is not improved over a number of generations, or a change in the problem environment is detected, or further exploration/exploitation of the solution space is required. Contents of the maintained memory can be either full-length representations, or randomly cut segments, or gene sequences randomly extracted from potentially promising solutions. A memory can be maintained either internally by incorporating accumulated experience into algorithmic parameters or externally as an archive to be used for experience-based guidance of the search procedure. A detailed review of memory-based strategies in genetic algorithms (GAs) is given in [15]. In order to give a clear idea on the two types memory organizations, two characteristic examples are presented below.

Montgomery et al. proposed the first internally implemented memory-based approach within ant colony optimization (ACO) [17]. The authors adapted the characteristic equations of ACO to include a weighting mechanism that represents the accumulated experience. The weighting is established on the nature of partial solutions reached in the current iteration. Elements showing a higher tendency toward better solutions are valued highly, whereas elements that lean to follow toward poorer solutions are weighed lower. The intention of the researchers of this approach is to present a more direct and objective feedback on the quality of the choices made, as well as to represent normal pheromone level and heuristic cost. It is claimed by the authors that the results attained for travelling salesman problem (TSP) instances are either equally well or better than those achieved using normal ant colony system algorithms.

Two population based external-memory approaches in ACO are introduced in [15,16]. In first of these approaches, variable-size solution segments taken from some elite individuals of previous iterations are stored in an external memory. In the second approach, partial permutation sequences extracted from a number of above average individuals are stored in the external memory. In both of the approaches, the memory is initially empty and a standard ACO algorithm runs for a small number of iterations to initialize the external partial solutions memory. Stored partial solutions are associated with their parents' objective function values that are used as measures for retrieval and updating the memory elements. Both of the approaches are used for the solution of QAP and significant improvements are achieved compared to conventional ACO implementations.

This paper is organized as follows: description of QAP and a list of metaheuristics employed for its solution are presented in Section 2. Section 3 presents a detailed explanation of GDA together with its implementation issues and different application areas. The tabu search metaheuristic used as a local search support within our proposed method is described in Section 4. Its algorithmic principles and various improvements are also given in this section. The proposed two-stage memory-powered and operator enhanced GDA approach for the solution of QAP is explained in Section 5. Lists of QAP benchmarks, their properties, best known fitness values and the experimental set up used for the solution of these problems are shown in Section 6. This section also covers detailed self and comparative evaluations of experimental results. Finally, Section 7 presents conclusions and the future work plans.

2. The quadratic assignment problem

The basic description of QAP is introduced by Koopmans and Beckmann in 1957 [18]. Given a set of N facilities, a set of N locations, distances between pairs of locations, and flows between pairs of facilities, QAP is described as the problem of assigning each facility to a particular location so as to minimize the sum of the product between flows and the distances. More formally, if $D=[d_{ij}]$ is the $N\times N$ distance matrix and $F=[f_{pq}]$ is the $N\times N$ flow matrix, where d_{ij} is the distance between locations i and j and f_{pq} is the amount of flow between facilities p and q, QAP can be described by the following equation:

$$\min_{\pi\in\Pi}\sum_{i=1}^{N}\sum_{j=1}^{N}d_{\pi(i)\pi(j)}f_{ij} \tag{1}$$

where Π is the set of all permutations of integers from 1 to N, and $\pi(i)$ gives the location of facility i within the current solution (permutation) $\pi\in\Pi$. The term $d_{\pi(i)\pi(j)}f_{ij}$ is the cost of simultaneously assigning facility i to location $\pi(i)$ and facility j to location $\pi(j)$.

Quadratic assignment problem belongs to the class of NP-hard combinatorial optimization problems and the largest instances that can be solved with exact algorithms are limited to instances of size around 30 [19–21]. Hence, the only feasible way to deal with the solution of large QAP instances is to use heuristic approaches which guarantee to reach a locally optimal solution in reasonable computation times. Several modern heuristics, mostly evolutionary or nature-inspired, are developed since 1980s and successfully applied for the solution of many provably hard optimization problems including the QAP. From 1980 until 2009, detailed reviews of heuristic and metaheuristic algorithms for QAP can be found in [21–23]. As can be seen in these well-prepared surveys, most widely referenced and successful metaheuristics for the QAP are simulated annealing (SA) [24,25], tabu search (TS) [26–29,25], genetic algorithms (GAs) [30–33], ant colony optimization (ACO), [34–36], greedy randomized adaptive search procedure (GRASP) [37,38], memetic algorithms (MAs) [39,40], hybrid methods (HMs) [41,42] and local search (LS) algorithms [43,44]. Recently, new QAP solution methodologies based on adaptation of powerful numerical optimization algorithms are presented. In this respect applications of particle swarm optimization (PSO) [45,46], differential evolution (DE) [47,48], imperialistic competitive algorithm (ICA) [49], and migrating birds optimization (MBO) algorithm [50] are reported to provide promising results compared to well-known metaheuristic algorithms for QAP. Among newly developed metaheuristic algorithms, chemical reaction optimization (CRO) has also been demonstrated to be a competitive algorithm for QAP and other difficult scheduling problems [51–54]. Furthermore, following the formulation of Geoffrion et al. [55], who formulated scheduling of parallel production lines as a QAP, different hard scheduling problems such as DAG scheduling problem can also be formulated as a QAP and solved using the algorithms mentioned above [56–58]. It should also be noted at this point that, the above mentioned metaheuristics exhibit almost similar performance for small-size and easy problem instances, whereas only those algorithms hybridized with a powerful local search method, such as tabu search, reach competitive scores for the difficult and large-size problem instances. Due to its experimentally proven efficiency for the QAP, we also used a variant tabu search, robust tabu search (RTS), algorithm as a local search support within the proposed GDA framework.

3. Great deluge algorithm

Great deluge algorithm is a trajectory based optimization algorithm that is similar to simulated annealing (SA) except for its dynamically adjusted level-based acceptance mechanism. The algorithm starts with a randomly constructed initial solution and has three fundamental parameters to be set initially. These are the estimated value of fitness for a globally optimal solution, the maximum number of iterations, and the initial value of the level parameter. Usually, the initial value of the level parameter is set equal to the fitness of the initial solution. Throughout the execution of GDA, the value of the level parameter is decayed linearly or nonlinearly, and the acceptance of new solutions depends on the level parameter. The basic GD algorithm is described in Algorithm 1.

Algorithm 1. Pseudocode of the basic great deluge algorithm.

$Iter \leftarrow 0$;
Build an initial solution, S_{Iter};
Compute the fitness of S_{Iter}, $F_{Old} \leftarrow F(S_{Iter})$;
$F_{Best} \leftarrow F_{Old}$;
Set the maximum number of iterations to *MaxIter*;
Estimate the fitness of a globally optimal solution, F_{Gbest};
Set the initial value of $Level_{Iter}$, $Level_{Iter} \leftarrow F_{Old}$;
Set the level decay parameter, $\Delta Level \leftarrow (F_{Old} - F_{Gbest})/MaxIter$;
Set Not improving length limit to *NILength*; /*This is one of termination conditions when the algorithm gets stuck at a locally optimal solution.*/
Set Not improving counter to zero, $NICount \leftarrow 0$;
$DONE \leftarrow 0$;
while(Not(DONE)),
  Generate a new solution, S_{Iter+1}, /* starting from S_{Iter}, using the available operators and the neighbourhood structure. */
  Compute the fitness of S_{Iter+1}, $F_{New} \leftarrow F(S_{Iter+1})$;
  If $F_{New} < F_{Best}$,
    $F_{Best} \leftarrow F_{New}$;
    $F_{Old} \leftarrow F_{New}$;
    $NICount \leftarrow 0$;
  else
    If $F_{New} \leq Level_{Iter}$,
      $F_{Old} \leftarrow F_{New}$;
      $NICount \leftarrow 0$;
    else
      $NICount \leftarrow NICount + 1$;
      If $NICount == NILength$,
        $DONE \leftarrow 1$;
      end if
    end if
  end if
  $Level_{Iter+1} \leftarrow Level_{Iter} - \Delta Level$;
  $Iter \leftarrow Iter + 1$;
end while

The fundamental distinctions between SA and GDA lies in the acceptance rule. While SA accepts new solutions stochastically based on Boltzmann distribution, GDA applies a deterministic rule using a dynamically changing parameter called *Level*. In SA, the magnitude of a uphill climbing move is controlled by the temperature parameter that effects the probability of being accepted from approximately all (when T is very large) to approximately none (when T is very small). The temperature is adjusted according to a predefined cooling schedule. In GDA, the magnitude of uphill climbing moves is controlled by the deterministic *Level* parameter such that all uphill moves with a magnitude less than the *Level* are accepted. The level parameter is adjusted according to a level decay mechanism. While SA is proved to be globally optimal under asymptotic thermal equilibrium conditions, no such proof exists for GDA. However, GDA is experimentally shown to be faster and more robust than SA for several hard optimization problems, like exam and course timetabling problems [2]. As already stated above, a two-stage memory powered and operator adapted GDA approach for QAP is presented in this paper. Details of its implementation and performance for well-known benchmark problem instances are presented in the following sections.

4. Tabu search algorithm

Tabu search (TS) metaheuristic is basically a greedy search algorithm with a memory of forbidden moves for the purpose of pushing the search towards yet undiscovered regions of the search space. This metaheuristic is first formalized by Glover [59–61] and several strategies to improve its efficiency were proposed later. A detailed review of well-known TS strategies can be found in [41]. The basic TS procedure starts with a complete feasible solution and all moves that are applicable (not tabu) on this solution are evaluated for their fitness improvements. Then, the move that is the best alternative for the time being is applied to generate a new solution within the neighbourhood of the parent solution. Consequently, the reverse move causing a return to the parent solution is defined as tabu, added to tabu list, and forbidden for a number of iterations. The number of iterations during which a move remains tabu is termed as its tabu tenure. Depending on the fitness landscape characteristics of the problem and tabu search trajectory within its solution space, a move can be removed from tabu list before the end of its tabu tenure. Considering the improvement provided by the best move, the tabu search procedure includes both downward (causing a decrease in fitness) and upward (causing an increase in fitness) moves. While the downward moves implement a steepest descent search throughout the fitness landscape and get stuck into locally optimal solutions, upward moves provide means for climbing hills and reaching other promising regions of the search space. A generic algorithm describing the basic tabu search procedure is given in Algorithm 2.

Algorithm 2. Basic tabu search procedure.

Tabu_Search(s_0,$f(s_0)$,*Max_Trials*,*Max_Failures*) % s_0 is the input solution.
  $DONE \leftarrow 0$;
  while (not DONE),
    Compute fitness changes of all moves that are applicable within the neighbourhood of s_0;
    Select the best move which is not tabu or it is tabu but satisfies all the aspiration criteria;
    Apply the move on s_0;
    Update $f(s_0)$;
    Update the tabu list and set tabu tenures of move elements;
    If fitness change caused by the best move is a strict improvement,
      Update *Best_Solution* found so far;
    else
      Num_Failures $\leftarrow$ *Num_Failures* + 1,
    end if
    Num_Trials $\leftarrow$ *Num_Trials* + 1;
    DONE=*Check_Termination*(*Num_Trials*, *Num_Failures*);
  end while

The tabu search variant used in our proposal is the robust tabu search algorithm (RTS) developed by Taillard [26]. Compared to the conventional TS algorithm and application to QAP, RTS introduces two basic improvements in terms of the tabu tenure parameter and the aspiration criteria. In this respect, tabu tenure of a move is selected randomly from a given range $[l_{min}, l_{max}]$. Tabu tenure is periodically changed after m iterations that is also chosen randomly from a given range. A move passes the aspiration criterion if both of the components exchanged by the move are not placed to locations which are already tried since the last t iterations.

Robust tabu search has successfully been applied for the solution of QAP either alone or as the local search partner of a hybrid metaheuristic. Algorithmic parameters of RTS considered in our experimental evaluations are given in Section 6.

5. The proposed two-stage memory powered GD algorithm

This study presents a novel methodology within a metaheuristic framework for the global optimization of a difficult combinatorial optimization problem, namely the QAP. Due to its NP-hard complexity, extraction of exact solutions for most of the QAP instances

are computationally infeasible and solution of this problem reduces to finding solutions of near-optimal quality within a countably finite but difficult fitness landscape. Among many approximation methods for optimization problems, metaheuristic algorithms are proved to be effective and computationally feasible alternatives. In fact, they are the best alternative when the trade-off between computational cost and solution quality is considered. Based on the number of potential solutions used in exploration of the solution space, metahueuristics for global optimization can be categorized as population- and trajectory-based algorithms. Trajectory-based algorithms use only one solution at a time to explore the solution space while the population-based methods are employing multiples of solutions for this purpose. The method presented in this paper uses a kind of hybrid approach that uses one solution in exploration of the solution space while two populations of high-quality and diversely distributed solutions are employed to efficiently direct the trajectory towards potentially promising regions of the solution space.

In the presented method, a solution vector X and a two-stage external memory $M = M_{FS} \cup M_{SS}$ are maintained, where M_{FS} and M_{SS} denote the first and the second stages of the external memory M, respectively. Elements of M are promising solutions extracted throughout the search process and the reasoning behind the two stage architecture is exploiting the useful information stored within diversely distributed good solutions without causing early convergence or being trapped into locally optimal solutions. In this respect, the first stage memory M_{FS} acts as a $short-term$ memory whose elements are changing frequently whenever a solution better than the worst of its elements is extracted, whereas the second stage memory M_{SS} functions as a $long-term$ memory whose components are updated only after a certain number of elements of M_{FS} is updated.

As mentioned above, elements of M_{FS} and M_{SS} are elite solutions found in previous iterations. First stage memory M_{FS} has a fixed size $|M_{FS}| = L$, whereas the maximum size of the second stage memory is $|M_{SS}| = K$, with the condition that $K \leq L$. That is, the number of elements in M_{SS} is initially K and may decrease up to one with increasing number of iterations due to convergence of solutions to one or more locally (globally) optimal solutions. Both memories are initially empty and M_{FS} is initialized with randomly built solutions. Then, to initialize M_{SS}, similarities between mutual elements of M_{FS} are computed and dissimilar elements of M_{FS} sorted in increasing order of their fitness values are inserted M_{SS}. As indicated before, the number of solutions inserted into M_{SS} can be at most K and initialization of the external memory M is completed this way. Based on our experimental observations, random initialization of M_{FS} results in a diverse distribution of its elements in the solution space and initial size of M_{SS} is almost always is equal to K.

After initialization of the two-stage external memory, the search procedure begins with a randomly built initial solution X, which is evaluated using the objective function $f(.)$ to get its fitness as $f(X)$. In order to implement a controlled diversification through the solution space and exploit the accumulated experience within the external memory, two different crossover operators and a mutation operator are applied probabilistically to convert the current solution X into X_{new}. The main objective of the two crossover operators is to direct the search towards promising regions of the search space without causing premature convergence to locally optimal solutions. The mutation operator, implemented through mutating a partially complete solution extracted from a randomly selected element of M_{SS}, aims controlled diversification around potentially promising solutions stored in M_{SS}. In this respect, probabilistic applications of the three search operators used throughout the proposed algorithm is described in Algorithm 3, where $randi([IMIN, IMAX])$ is the function generating uniformly distributed integer numbers in $[IMIN, IMAX]$, $rand()$ is the function generating uniform random numbers in $[0, 1]$, p_1 and p_2 are the parameters determining the frequency of selection for each of the three moves. Values of p_1 and p_2 are determined experimentally.

Algorithm 3. Probabilistic selection of the three search operators (moves).

```
for i=1 to N,
  R1 ← rand();
  r ← randi([1, |MSS|]);
  if R1 ≤ p1,
    R2 ← rand();
    if R2 ≤ p2,
      r' ← randi([1, |MSS|]); /* */
      Xnew ← Xover1(MSS(r, :), MSS(r', :));
    else
      Xnew ← Mutation(MSS(r, :));
    end if
  else
    Xnew ← Xover2(X, MSS(r, :));
  end if
endfor
```

The first crossover operator is the swap-path crossover (SPC) that was originally developed by Glover and named as path-relinking operator [63]. SPC is one of the very effective operators for permutation-type problems and it is described as follows [62]: let Par_1 and Par_2 be two parents to be crossed over, c_1 and c_2 be two different uniformly randomly selected crossover points such that $1 \leq c_1 \leq c_2 \leq N$, and N be the chromosome length that is equal to the number of facilities. Then, the allelic values of the two parents are examined from c_1 to c_2; if the allelic values of the two parents at a locus are not the same, they are interchanged between Par_1 and Par_2 so that the allelic of values of the two parents at the current position become the same. An example illustrating the implementation of SPC is presented in Fig. 1.

The second move is a mutation operator that was already used in [36] within a memory based ACO framework and found successful in reaching high quality solutions for difficult quadratic assignment problems. In the execution of this operator, first a partial permutation of facilities is extracted from a randomly selected element of the second stage archive M_{SS}. An important property of this partial permutation is that its assigned facilities share the same locations with a promising solution of the second stage archive the elements of which are dissimilar to each other according to a predefined similarity measure. The unassigned facilities of this partial permutation are assigned to empty locations through uniformly random selection. Fig. 2 illustrates the implementation of this mutation operator.

The third move is a crossover operator taking the current solution and a randomly retrieved element of M_{SS} as its arguments. This crossover operator is implemented in [64] and experimentally shown to be an effective search operator for the multiobjective version of QAP. Considering the current solution X and a randomly retrieved element of M_{SS}, the segment based crossover operator first cuts a randomly located and random length segment from the archive element. Then, this cut segment is pasted over the same positional segment of the current solution X. Elements of the pasted segment that are repeated outside the cut region are replaced by those elements of the current solution previously located in the same area. This replacement is done in the order that the replaced facilities appear in the current solution X. An example demonstrating the application of this operator is given in Fig. 3.

Together with the above described memory-based search operators, another novelty brought by the proposed method is the structure and organization of the maintained two-stage external memory. The two-stage architecture allowing the exploitation of accumulated search experience without causing premature convergence works as follows: as explained before, the fixed-size first-stage memory M_{FS} is initially empty and it is filled with randomly constructed solutions. Elements of M_{FS} are updated every

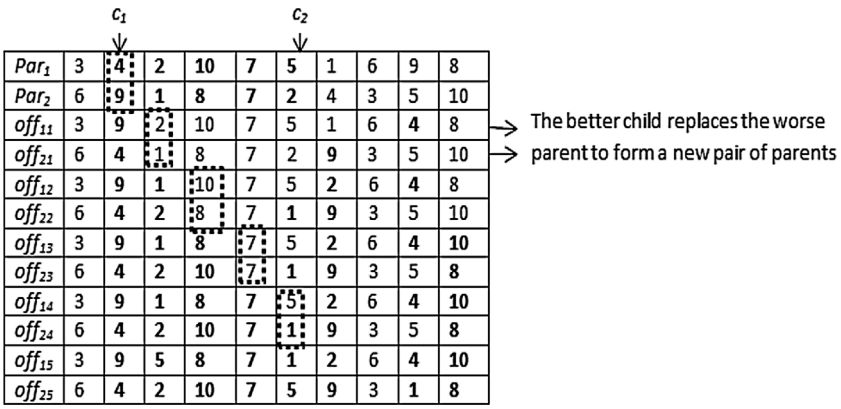

Fig. 1. An illustration of the swap path crossover operator.

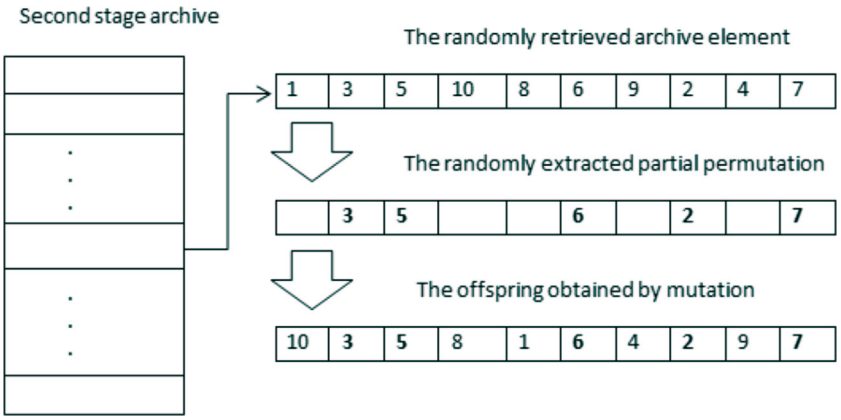

Fig. 2. Implementation of the partial permutation based mutation operator.

time a new solution better than its worst element is discovered. In such a case, the worst element of M_{FS} is replaced by the new better solution. In order to speed up the removal and insertion procedures, elements of M_{FS} are always kept sorted in increasing order of their fitness values. Consequently, elements of M_{SS} are determined as follows: since elements of M_{SS} are required to be potentially promising solutions which are diversely distributed within the solution space, solutions in M_{FS} are checked for variable-space similarity and those elements with similarity scores lower than a predefined threshold λ are labelled as *similar*. Then, dissimilar elements of M_{FS} are sorted in increasing order of their fitness values and inserted into M_{SS} in order without exceeding its maximum size, K. Diverse distribution and dissimilarity of elements in M_{SS} aims to prevent premature convergence due to initially superior but locally optimal solutions and to explore the solution space around the several promising solutions. Depending on the defined similarity threshold and the current iteration of the optimization procedure, the number of elements in M_{SS} changes between 1 and K. The distance based variable-space similarity measure used in the proposed approach is the Hamming distance that is defined as the number of locations at which two permutations are different.

$$Sim(X, Y) = Hamming\_Distance(X, Y) \tag{2}$$

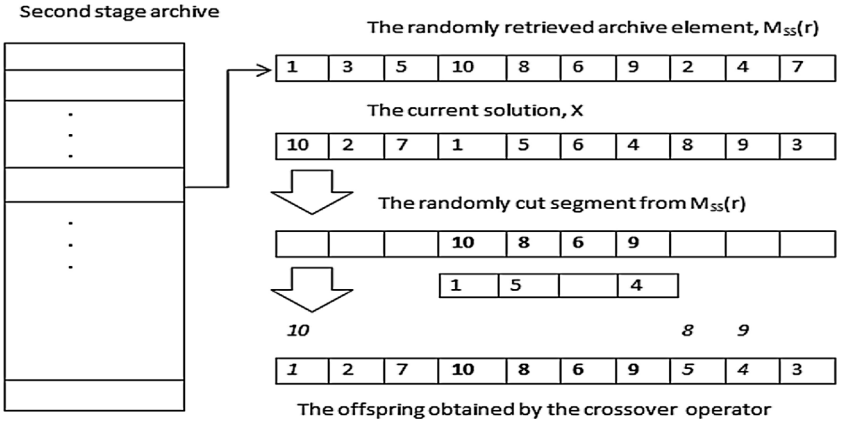

Fig. 3. Implementation of the partial segment based crossover operator.

The $Sim(X, Y)$ score is taken as a measure of closeness between the two vectors X and Y. From the above definition, it is clear that $Sim(Y, X) = Sim(X, Y)$. and when $Sim(X, Y)$ is less than a predefined threshold λ, then the two vectors are classified as *similar* and one of them is marked not to be insert it into M_{SS}. This way, all elements of M_{SS} are kept maximally apart from each other. This similarity based evaluation of solution vectors in variable space (not in fitness space) allows diverse sampling of promising solutions within the solution space. Hence, representative solutions from different localities allow simultaneous intensification around more than one solution, even when their fitness values are close to each other. Elements of M_{SS} are updated as follows: when elements of M_{FS} are updated for a predefined q number of times, the total external memory M is built as $M = M_{FS} \cup M_{SS}$. Obviously, the size of M, $|M| = |M_{FS}| + |M_{SS}|$, depends on the iteration at which the union is made because of the iteration dependant size of M_{SS}. Consequently, similarity measure between mutual components of M are calculated using Eq. (2) and only one of those elements that are similar to each other remains unmarked. Unmarked elements of M are sorted in increasing order of their fitness values and they are then inserted into M_{SS} without exceeding its maximum limit. This combined evaluation and update methodology for M_{SS} enables the storage of representative solutions from multiple promising regions of the solution space and searching around them until they are replaced by better representative solutions. Hence, the proposed architecture with variable-space similarity evaluation is quite suitable for fitness landscapes with multiple optimal solutions with the same level of fitness values.

As pointed out before, for hard permutation-type combinatorial optimization problems like QAP, most of the metaheuristic methods need a local search support to reach competitively good solutions within reasonable computation times. In this respect, tabu search is one of the best local search methods used for the solution of combinatorial optimization problems for which hybridization with metaheuristics is experimentally shown to reach the best found solutions over benchmark problem instances. Accordingly, based on the success of different tabu search variants over the QAP instances, we used the robust tabu search (RTS) that is described in Section 4. However, as shown in Algorithm 5, our use of RTS as a local search support is not as frequent as the other hybrid approaches. RTS procedure is used only after M_{SS} is updated and it is used for the improvement of a few randomly selected individuals of M_{SS} that stores the most promising and diversely located solutions extracted so far.

Based on the above explanations, an algorithmic description of the proposed algorithm is presented in Algorithms 3 and 4. While Algorithm 3 introduces the steps of initializing the starting solution and search procedure parameters, Algorithm 4 describes the fundamental computational details of the proposed method.

Algorithm 4. Initialization of starting solution and search algorithm parameters.

$Iter \leftarrow 0$,
Build an initial solution, S_{Iter};
Compute the fitness of S_{Iter}, $F_{Old} \leftarrow F(S_{Iter})$;
$F_{Best} \leftarrow F_{Old}$;
Set the maximum number of iterations to $MaxIter$;
Estimate the fitness of a globally optimal solution, F_{Gbest};
Set the initial value of $Level_{Iter}$, $Level_{Iter} \leftarrow F_{Old}$;
Set the level decay parameter $\Delta Level \leftarrow (F_{Old} - F_{Gbest})/MaxIter$;
Set "not improving length limit" to $NILength$; /*This is the conditions when the algorithm reinitializes the current solution when it gets stuck at a locally optimal solution.*/
Set the M_{SS} update indicator to q;
Set the "similarity threshold" to λ;
Set "not improving counter" to zero, $NICount \leftarrow 0$;
Set M_{SS} update counter to zero, $MSSUpCounter \leftarrow 0$;
Set $|M_{FS}| \leftarrow L$, $Max\_M_{SS}\_Size \leftarrow K$;
Initialize M_{FS} with randomly built solutions and compute fitness values of solutions in M_{FS};
Set $M_{SS} \leftarrow \emptyset$;
$Similarity\_Matrix$=Compute mutual similarities between elements of M_{FS} (Equation (2));
for each row i of $Similarity\_Matrix$,
  if the i^{th} element is not marked as similar before,
    Mark all solution with a similarity measure below λ
    as *similar* to the i^{th} solution;
  end if
end for
Sort unmarked elements of M_{FS} in increasing order of their fitness values;
Insert unmarked elements of M_{FS} into M_{SS} without exceeding its maximum size;

Algorithm 5. Pseudocode of the two-stage external memory based great deluge algorithm.

Iniatilize S_{Iter} and Other Search Parameters Using Algorithm 3;
$DONE \leftarrow 0$;
while(Not(DONE));
  Generate a new solution, S_{Iter+1}, starting from S_{Iter}; (Algorithm 2)
  Compute the fitness of S_{Iter+1}, $F_{New} \leftarrow F(S_{Iter+1})$;
  if $F_{New} < F(M_{FS}(L))$;
    Remove L^{th} (i.e. the worst) element of M_{FS};
    Insert $_{Iter+1}$ into M_{FS}; (its positions depends on its fitness value)
    $MSSUpCounter \leftarrow MSSUpCounter + 1$;
    if $MSSUpCounter > q$,
      $M \leftarrow M_{FS} \cup M_{SS}$;
      $Similarity\_Matrix$=Compute mutual similarities between elements of M;
      for each row i of $Similarity\_Matrix$,
        if the i^{th} element is not marked as similar before,
          Mark all solution with a similarity measure below λ
            as *similar* to the i^{th} solution;
        end if
      end for
      Sort unmarked elements of M_{FS} in increasing order of their fitness values;
      Insert unmarked elements of M_{FS} into M_{SS}
    without exceeding its maximum size;
      $MSSUpCounter \leftarrow 0$;
    end if
    Apply local search RTS for $max(1, \lceil 0.05 * N \rceil)$ randomly selected elements of M_{SS} and insert the found solutions into M_{SS};
  end if
  if $F_{New} < F_{Best}$,
    $F_{Best} \leftarrow F_{New}$;
    $F_{Old} \leftarrow F_{New}$;
    $NICount \leftarrow 0$;
  else
    if $F_{New} \leq Level_{Iter}$,
      $F_{Old} \leftarrow F_{New}$; $NICount \leftarrow 0$;
    else
      $NICount \leftarrow NICount + 1$;
      if $NICount == NILength$,
        Initialize S_{Iter+1} randomly and evaluate its fitness value;
        $NICount \leftarrow 0$;
      end if
    end if
  end if
  $Level_{Iter+1} \leftarrow Level_{Iter} - \Delta Level$;
  $Iter \leftarrow Iter + 1$;
  $DONE \leftarrow CheckTermination(MaxIter, NICount)$;
end while

6. Experimental studies

To evaluate the performance of the proposed algorithm and exhibit its comparative success against well-known metaheuristics, 137 benchmark problems taken from QAPLIB [65], 21 instances named as *taixxeyy* [66], and the 12 Drezner instances [21,67] are considered. In [43], QAPLIB problems are categorized into four types depending on properties of their flow and distance matrices as follows:

I. *Unstructured, randomly generated instances* for which the elements of distance an flow matrices are generated uniformly randomly within a range of integer values (e.g. **taixxa** instances). These instances are among the most difficult ones

in terms of the computational time required to find a solution closest to the best known one. Table 1 lists the properties of QAP instances in this category where BKS denotes the fitness value of the best known solution (BKS). It is stated in [43] that, in the limit of large instance sizes, the difference between the upper and lower bounds on the optimal fitness values converges to zero. This implies that these instances have a unique or a very small set of optimal solutions.

II. *Instances with grid based distance matrices* for which the locations are assumed to be on the nodes of an $n_1 \times n_2$ grid and the distances between locations are taken as Manhattan distances between grid nodes (e.g. **nugxx** and **skoxx** instances). These problem instances are illustrated in Table 2.

III. *Real-life instances* stemming from applications of QAP with flow matrices having many zero entries (e.g. **chrxxx** and **escxxx** instances). These instances are generally small in size and easy to solve for any powerful metaheuristic with an appropriate local search support. A list of these problem instances are given in Table 3.

IV. *Randomly generated real-life like instances* having flow matrices similar to real-life instances with the exception that the nonzero entries of flow matrices are uniformly distributed within a predefined range of integer values (e.g. **taixxb** instances). Table 4 shows list of these problem instances.

We also considered two sets of more difficult QAP benchmark instances, namely, the Taillard's *taixxeyy* instances and the Drezner's *drexx* instances. Unlike the *taixxa* instances, the *taixxeyy* instances are characterized as structured and symmetric such that certain facility permutations (exchanges) do not change the fitness value. That is, fitness-equivalent solutions in the form of different permutations are visited again and again through out the exploration process that makes these QAP instances hard for metaheuristic algorithms. *Taixxeyy* instances of size 27 and 45 are solved optimally by the proposed method in all trials. The average *cpu times* for *tai27eyy* and *tai45eyy* instances are 0.27 and 17.59 min, respectively. Best known fitness values of *tai75eyy* instances and *tai125e01* instance are available in Table 6. Comparative evaluations of the proposed method for these difficult QAP instances are also given in the following subsections.

Considering the Drezner instances, they are experimentally shown to be hard QAP instances for metaheuristic methods. In these problems, locations are assumed to be distributed over nodes of a rectangular grid with highly sparse non-zero flow values such that the objective function will change only for a few pair-wise exchanges of facilities and, for those pair-wise facility exchanges causing a change, the fitness function changes quite steeply. This also means that it is hard to reach an optimal solution from its neighbourhood due to very small neighbourhood size and sharply changing fitness landscape [21,67]. Table 7 presents characteristics of Drezner problem instances.

In experimental studies, except the *taixxeyy* instances, each benchmark problem is solved in 10 consecutive independent runs with the termination criteria set in terms of maximum $20\,000 \times N$ (N = number of facilities) fitness evaluations for all the benchmark problems and for all the algorithms in the experimental suit. Experiments for *taixxeyy* instances are conducted 20 times for the purpose of compatibility with the only one reference with results of these instances. Algorithmic parameters of all algorithms are kept the same for all the benchmark problems and no interactive intervention is made throughout the individual runs. All implementations are carried out using *Matlab*® 10 programming language environment and toolboxes, and a personal PC with 8 GB main memory and 2.1 GHz clock speed.

The proposed algorithm contains two sets of parameters to be set as follows: The first set contains the three parameters of the great deluge algorithm as described in Algorithm 1. These are $Level_{Iter}$, $\Delta Level$ and not improving length limit *NILength* parameters. Values used for *Level* and $\Delta Level$ parameters are given in Algorithm 3, and settings for all the other parameters mentioned so far are listed in Table 1. Tabu search aspiration criteria are set as described in [26] and all parameter values are globally used for all executions and for all problem sets.

6.1. Self-performance evaluations

Experiments with QAPLIB problems started with unstructured and uniformly randomly generated type-I instances. As explained above, these are among the hardest problem instances for metaheuristics due to fact that they have unique or very small sets of optimal solutions. Table 2 illustrates the average, best, and worst percent deviations (APD, BPD, WPD, respectively) from the fitness of best known solutions (BKS) where the subscripts in parentheses show the number of times BKS is found out of the 10 runs. All cpu-times are in minutes. It can be seen that the best known solutions are found for most of the problem instances. Among the 28 problems in this category, BKS could not be extracted for only 7 difficult instances. For the difficult problems, from *tai40a* to *taii100a* and from *lipa70a* to *lipa90a*, the APD values are all less than 1.0%, BPD values are all less than 0.4%, and WPD values are all less than 1.5% of BKS. As will be illustrated in the following subsections, majority of the APD values are better than those of RoTS. Considering the cpu-times, they are within the feasible limits, the largest cpu-time is spent for the solution of *tai100a* instance for which each trial took around 2 cpu-hours.

The second category of QAPLIB problems, with grid-based distances between locations, includes both easy (*nugxx*) and difficult (*skoxx*) instances. Experimental results for this category of problems are given in Table 3. Among the 36 QAP problems in this category, only two of them (*sko100f* and *tho150*) could not be solved optimally. All *nugxx* instances are solved optimally in all trials and the cpu-time required for each trial is less than two minutes for the maximum size *nug30* instance. The hardest problems of this category are the *skoxx* instances of size 49 and above. For the 12 *sko49-sko100f* instances, only *sko100f* could not be solved optimally. APD value for this problem is 0.013% which shows that the average performance of the proposed method for this difficult instance is very close to optimality. On the other hand, the WPD value for *sko100f* instance over the 10 trials is 0.262% which is another indication on the effectiveness of the proposed method; the closeness of the worst fitness value to BKS is less than 0.3%. Another problem that could not be solved optimally is the large size *tho150* instance. The APD and WPD values for this problem are 0.039% and 0.139%, respectively. The two-stage memory architecture that enables exploration around multiple promising solutions achieved the relative percent quality of the worst solution for this difficult instance is less than 0.15%. Cpu-times are around 2.5 hours for *sko100x* instances and the *will100* instance, while it is 7.5 hours for the *tho150* instance.

Tables 4 and 5 illustrate the results for simple real-life instances with flow matrices having many zero entries. The proposed method found optimal solutions for all problems in this category within short computation times. The largest cpu-times are spent for the solution of *ste36a, esc32a* and *esc128* for which the optimal solutions are found in approximately 2 minutes. For majority of other instances in this category, optimal solutions are reached in less than 0.5 minutes.

Experimental results for the last set of QAPLIB instances, randomly generated real-life like instances, with flow matrices containing nonzero entries uniformly distributed within a predefined range of integer values are given in Table 6. Compared to *taixxa* instances, these *taixxb* instances are simpler and they are solved

Table 1
Values of algorithmic parameters.

NILength	L	K	q	λ	p_1	p_2	Tabu tenure interval	Tabu search fitness evals.
100	N	N/2	5*N	max(5,0.2*N)	0.75	0.25	[0.9*N,1,1*N]	10*N

Table 2
Experimental scores with respect to APD, BPD, WPD and CPU time for Type-I QAPLIB benchmark problems.

Problem	Size	BKS	APD	BPD	WPD	Time (min)
tai20a	20	703482	$0.0_{(10)}$	0.0	0.0	2.096
tai25a	25	1167256	$0.0_{(10)}$	0.0	0.0	15.820
tai30a	30	1818146	$0.091_{(6)}$	0.0	0.174	20.291
tai35a	35	2422002	$0.153_{(2)}$	0.0	0.267	24.991
tai40a	40	3139370	0.261	0.125	0.343	27.783
tai50a	50	4938796	0.276	0.198	0.376	41.139
tai60a	60	7205962	0.448	0.218	0.639	78.862
tai80a	80	13515450	0.832	0.289	0.921	111.337
tai100a	100	21054656	0.874	0.337	1.347	138.321
rou12	12	235528	$0.0_{(10)}$	0.0	0.0	0.022
rou15	15	354210	$0.0_{(10)}$	0.0	0.0	0.044
rou20	20	725522	$0.0_{(10)}$	0.0	0.0	1.716
lipa20a	20	3683	$0.0_{(10)}$	0.0	0.0	0.044
lipa20b	20	27076	$0.0_{(10)}$	0.0	0.0	0.033
lipa30a	30	13178	$0.0_{(10)}$	0.0	0.0	0.125
lipa30b	30	151426	$0.0_{(10)}$	0.0	0.0	0.057
lipa40a	40	31538	$0.0_{(10)}$	0.0	0.0	0.682
lipa40b	40	476681	$0.0_{(10)}$	0.0	0.0	0.107
lipa50a	50	62093	$0.0_{(10)}$	0.0	0.0	3.901
lipa50b	50	1210244	$0.0_{(10)}$	0.0	0.0	4.092
lipa60a	60	107218	$0.0_{(10)}$	0.0	0.0	22.064
lipa60b	60	2520135	$0.0_{(10)}$	0.0	0.0	1.909
lipa70a	70	169755	$0.046_{(4)}$	0.0	0.061	34.877
lipa70b	70	4603200	$0.0_{(10)}$	0.0	0.0	1.759
lipa80a	80	253195	0.055	0.042	0.065	49.877
lipa80b	80	7763962	$0.0_{(10)}$	0.0	0.0	3.700
lipa90a	90	360630	0.157	0.049	0.253	91.442
lipa90b	90	12490441	$0.0_{(10)}$	0.0	0.0	9.774

Table 3
Experimental scores with respect to APD, BPD, WPD and CPU time for type-II QAPLIB benchmark problems.

Problem	Size	BKS	APD	BPD	WPD	Time (min)
nug12	12	578	$0.0_{(10)}$	0.0	0.0	0.013
nug14	14	1014	$0.0_{(10)}$	0.0	0.0	0.021
nug15	15	1150	$0.0_{(10)}$	0.0	0.0	0.033
nug16a	16	1610	$0.0_{(10)}$	0.0	0.0	0.025
nug16b	16	1240	$0.0_{(10)}$	0.0	0.0	0.029
nug17	17	1732	$0.0_{(10)}$	0.0	0.0	0.040
nug18	18	1930	$0.0_{(10)}$	0.0	0.0	0.058
nug20	20	2570	$0.0_{(10)}$	0.0	0.0	0.068
nug21	21	2438	$0.0_{(10)}$	0.0	0.0	0.091
nug22	22	3596	$0.0_{(10)}$	0.0	0.0	0.038
nug24	24	3488	$0.0_{(10)}$	0.0	0.0	0.103
nug25	25	3744	$0.0_{(10)}$	0.0	0.0	0.088
nug27	27	5234	$0.0_{(10)}$	0.0	0.0	0.129
nug28	28	5166	$0.0_{(10)}$	0.0	0.0	0.382
nug30	30	6124	$0.0_{(10)}$	0.0	0.0	1.783
sko42	42	15812	$0.0_{(10)}$	0.0	0.0	2.242
sko49	49	23386	$0.005_{(6)}$	0.0	0.032	3.849
sko56	56	34458	$0.001_{(8)}$	0.0	0.025	14.721
sko64	64	48498	$0.0_{(10)}$	0.0	0.0	29.385
sko72	72	66256	$0.007_{(2)}$	0.0	0.072	37.992
sko81	81	90998	$0.019_{(1)}$	0.0	0.085	57.141
sko90	90	115534	$0.031_{(2)}$	0.0	0.182	93.826
sko100a	100	152002	$0.029_{(1)}$	0.0	0.188	153.173
sko100b	100	153890	$0.015_{(1)}$	0.0	0.124	164.272
sko100c	100	147862	$0.013_{(1)}$	0.0	0.201	154.514
sko100d	100	149576	$0.017_{(1)}$	0.0	0.234	148.855
sko100e	100	149150	$0.016_{(6)}$	0.0	0.260	146.145
sko100f	100	149036	0.013	0.006	0.262	153.383
scr12	12	31410	$0.0_{(10)}$	0.0	0.0	0.013
scr15	15	51140	$0.0_{(10)}$	0.0	0.0	0.047
scr20	20	110030	$0.0_{(10)}$	0.0	0.0	0.138
tho30	30	149936	$0.0_{(10)}$	0.0	0.0	5.034
tho40	40	240516	$0.003_{(8)}$	0.0	0.006	14.024
tho150	150	8133398	0.039	0.013	0.139	512.787
wil50	50	48816	$0.002_{(9)}$	0.0	0.004	18.811
wil100	100	273038	$0.008_{(1)}$	0.0	0.017	155.128

Table 4
Experimental scores with respect to APD, BPD, WPD and CPU time for type-III QAPLIB benchmark problems.

Problem	Size	BKS	APD	BPD	WPD	Time (min)
bur26a	26	5426670	$0.0_{(10)}$	0.0	0.0	0.243
bur26b	26	3817852	$0.0_{(10)}$	0.0	0.0	0.499
bur26c	26	5426795	$0.0_{(10)}$	0.0	0.0	0.154
bur26d	26	3821225	$0.0_{(10)}$	0.0	0.0	0.487
bur26e	26	5386879	$0.0_{(10)}$	0.0	0.0	0.126
bur26f	26	3782044	$0.0_{(10)}$	0.0	0.0	0.193
bur26g	26	10117172	$0.0_{(10)}$	0.0	0.0	0.187
bur26h	26	7098658	$0.0_{(10)}$	0.0	0.0	0.107
chr12a	12	9552	$0.0_{(10)}$	0.0	0.0	0.155
chr12b	12	9742	$0.0_{(10)}$	0.0	0.0	0.080
chr12c	12	11156	$0.0_{(10)}$	0.0	0.0	0.185
chr15a	15	9896	$0.0_{(10)}$	0.0	0.0	0.261
chr15b	15	7990	$0.0_{(10)}$	0.0	0.0	0.249
chr15c	15	9504	$0.0_{(10)}$	0.0	0.0	0.269
chr18a	18	11098	$0.0_{(10)}$	0.0	0.0	0.352
chr18b	18	1534	$0.0_{(10)}$	0.0	0.0	0.354
chr20a	20	2192	$0.0_{(10)}$	0.0	0.0	0.649
chr20b	20	2298	$0.0_{(10)}$	0.0	0.0	0.840
chr20c	20	14142	$0.0_{(10)}$	0.0	0.0	0.588
chr22a	22	6156	$0.0_{(10)}$	0.0	0.0	0.853
chr22b	22	6194	$0.0_{(10)}$	0.0	0.0	0.897
chr25a	25	3796	$0.0_{(10)}$	0.0	0.0	0.972
els19	19	17212548	$0.0_{(10)}$	0.0	0.0	0.088
had12	12	1652	$0.0_{(10)}$	0.0	0.0	0.020
had14	14	2724	$0.0_{(10)}$	0.0	0.0	0.005
had16	16	3720	$0.0_{(10)}$	0.0	0.0	0.012
had18	18	5358	$0.0_{(10)}$	0.0	0.0	0.007
had20	20	6922	$0.0_{(10)}$	0.0	0.0	0.006

Table 5
Experimental scores with respect to APD, BPD, WPD and CPU time for type-III QAPLIB benchmark problems (continued).

Problem	Size	BKS	APD	BPD	WPD	Time (min)
kra30a	30	88900	$0.0_{(10)}$	0.0	0.0	0.947
kra30b	30	91420	$0.0_{(10)}$	0.0	0.0	1.392
kra32	32	88700	$0.0_{(10)}$	0.0	0.0	0.626
ste36a	36	9526	$0.0_{(10)}$	0.0	0.0	2.173
ste36b	36	15852	$0.0_{(10)}$	0.0	0.0	0.406
ste36c	36	8239110	$0.0_{(10)}$	0.0	0.0	1.395
esc16a	16	68	$0.0_{(10)}$	0.0	0.0	0.003
esc16b	16	292	$0.0_{(10)}$	0.0	0.0	0.001
esc16c	16	160	$0.0_{(10)}$	0.0	0.0	0.004
esc16d	16	16	$0.0_{(10)}$	0.0	0.0	0.001
esc16e	16	28	$0.0_{(10)}$	0.0	0.0	0.002
esc16f	16	0	$0.0_{(10)}$	0.0	0.0	0.046
esc16g	16	26	$0.0_{(10)}$	0.0	0.0	0.002
esc16h	16	996	$0.0_{(10)}$	0.0	0.0	0.001
esc16i	16	14	$0.0_{(10)}$	0.0	0.0	0.001
esc16j	16	8	$0.0_{(10)}$	0.0	0.0	0.001
esc32a	32	130	$0.0_{(10)}$	0.0	0.0	2.209
esc32b	32	168	$0.0_{(10)}$	0.0	0.0	0.598
esc32c	32	642	$0.0_{(10)}$	0.0	0.0	0.005
esc32d	32	200	$0.0_{(10)}$	0.0	0.0	0.026
esc32e	32	2	$0.0_{(10)}$	0.0	0.0	0.004
esc32f	32	2	$0.0_{(10)}$	0.0	0.0	0.004
esc32g	32	6	$0.0_{(10)}$	0.0	0.0	0.005
esc32h	32	438	$0.0_{(10)}$	0.0	0.0	0.005
esc64a	64	116	$0.0_{(10)}$	0.0	0.0	0.038
esc128	128	64	$0.0_{(10)}$	0.0	0.0	2.135

Table 6
Experimental scores with respect to APD, BPD, WPD and CPU time for type-IV QAPLIB benchmark problems.

Problem	Size	BKS	APD	BPD	WPD	Time (min)
tai20b	20	122455319	$0.0_{(10)}$	0.0	0.0	0.074
tai25b	25	344355646	$0.0_{(10)}$	0.0	0.0	0.3269
tai30b	30	637117113	$0.0_{(10)}$	0.0	0.0	1.148
tai35b	35	283315445	$0.0_{(10)}$	0.0	0.0	6.387
tai40b	40	637250948	$0.0_{(10)}$	0.0	0.0	4.877
tai50b	50	458821517	$0.0_{(10)}$	0.0	0.0	10.249
tai60b	60	608215054	$0.005_{(5)}$	0.0	0.014	33.639
tai64c	64	1855928	$0.0_{(10)}$	0.0	0.0	0.005
tai80b	80	818415043	0.025	0.004	0.051	72.601
tai100b	100	1185996137	0.028	0.011	0.047	138.533
tai150b	150	498896643	0.051	0.023	0.069	257.967

Table 7
Experimental scores with respect to APD, BPD, WPD and CPU time for the Tai75eyy and Tai125e01 instances.

Problem	Size	BKS	APD	BPD	WPD	Time (min)
Tai75e01	75	14488	$4.097_{(8)}$	0.856	5.914	91.074
Tai75e02	75	14444	$4.165_{(8)}$	1.018	6.055	90.451
Tai75e03	75	14154	$1.081_{(14)}$	0.194	2.171	73.331
Tai75e04	75	13694	$1.005_{(16)}$	0.183	1.978	75.227
Tai75e05	75	12884	$1.412_{(11)}$	0.242	3.781	79.431
Tai75e06	75	12534	$2.115_{(10)}$	1.066	3.113	87.441
Tai75e07	75	13782	$4.662_{(7)}$	0.732	4.926	93.773
Tai75e08	75	13948	$3.951_{(8)}$	1.131	5.009	88.665
Tai75e09	75	12650	$1.051_{(15)}$	0.201	4.329	76.305
Tai75e10	75	14192	$2.910_{(10)}$	1.141	4.956	88.803
Tai75e11	75	15250	$1.021_{(16)}$	0.113	2.978	74.741
Tai75e12	75	12760	$4.619_{(7)}$	1.551	5.511	79.965
Tai75e13	75	13024	$2.922_{(11)}$	1.109	5.963	83.783
Tai75e14	75	12604	$1.601_{(12)}$	0.305	3.003	76.076
Tai75e15	75	14294	$1.007_{(16)}$	0.156	2.709	77.877
Tai75e16	75	14204	$1.081_{(14)}$	0.139	3.313	75.095
Tai75e17	75	13210	$1.223_{(12)}$	0.173	3.751	78.934
Tai75e18	75	13500	$3.033_{(9)}$	1.216	6.117	81.816
Tai75e19	75	12060	$1.044_{(15)}$	0.157	2.904	76.093
Tai75e20	75	15260	$1.374_{(14)}$	0.207	6.935	73.526
tai125e01	125	35450	7.751	4.983	11.278	230.689

optimally up to size 64. For problem size greater than or equal to 80, optimal solutions could not be extracted, however APD, BPD and WPD values listed in Table 6 indicate that the solutions found by the proposed method are very close to BKSs. The largest size instance in this category is *tai150b* for which the APD, BPD, and WPD values are 0.051%, 0.023% and 0.069%, respectively. All the three indicators are below 0.07% even for this very large size QAP instance.

Considering the difficult *taixxeyy* instances, all *tai25eyy* and *tai45eyy* instances are solved optimally in all 20 trials. Running time information for these problems was given above. Since the best known fitness values are for *Tai75eyy* and *tai125e01* were available in [68], these instances are considered in our experimental results for both self and comparative evaluations. Table 7 presents the APD, BPD, WPD and cpu-time scores for these problem instances. The proposed method extracted optimal solutions for all of the 20 *tai75eyy* instances and the average cpu-time required to complete one experimental trial was around 1.5 h. Further evaluations with respect to other algorithms attempted to solve these QAP instances are given below.

The last set of QAP instances considered in our experimental evaluations contains the difficult *drexx* instances. The proposed method solved these instances optimally up to size 42. Table 8 illustrates the scores for *drexx* instances for which the proposed method could not locate optimal solutions for sizes 56 and above, and deviation BKSs increase with increasing problem sizes. Even though the proposed method could not locate BKSs for some of *drexx* instances, as shown in Table 14, the APD scores achieved are much better than its single competitor.

6.2. Comparative performance evaluations

Tables 9–14 exhibit the comparative scores of the proposed algorithm against well-known metaheuristics that are frequently referred to in literature. Scores associated with all metaheursitics are taken from [29] and verified over the individual references. Abbreviations used for all algorithms are the same as the ones used in [29] and their corresponding references from which the associated scores are verified are as follows:

1 DivTS: DivTS restart tabu search approach [29];
2 RTS: Robust tabu search [26,29];
3 GRASP: Greedy randomized adaptive search procedure [37];
4 ACO-GA/LS: A hybrid metaheuristic for QAP [69];
5 ETS1: A tabu search algorithm for QAP [28];
6 GS/TS: GAs hybridized with a ruin and recreate procedure [31];
7 GA/TS/I: An Improved hybrid GA [32];
8 GA-S/TS: GA hybrid with standard TS [30];
9 GA-C/TS: GA hybrid with concentric TS [30];
10 GA/IC-TS: GA hybrid with improved concentric TS [27];
11 I-ILS6: Iterarted local search for QAP [43];
12 ACO2, ACO3: ACO algorithms for QAP [34].

Tables below illustrate the APD or BPD scores, whichever is available, for the proposed method and all of its competitors listed above. The proposed method is named as TMSGD-QAP from now on in all tables. It should be noted at this point that, experimental settings, programming platforms and hardware resources of almost all algorithms considered for comparative evaluations are different, hence a true comparison under the same experimental conditions is not possible. Meaningful runtime comparisons are not usually possible also for the same reasons. However, as can be seen from the tables on self-evaluation, cpu-times of TMSGD-QAP for all benchmark problems are within feasible limits compared to the run-times reported by its competitors. For example, run-times reported for RTS and DivTS in [29] are very close to the ones presented in the self-evaluation tables given above. Even if the computational platforms are not exactly the same, this observation can be seen as an indicator on the feasibility of our cpu-time scores. In addition to these, another obstacle making the true comparison impossible is the unavailability of results for all benchmark problems for all algorithms. Blank spaces in table entries indicate the absence of scores between the corresponding problem and metaheuristic entries. These are the main reasons of comparing deviations from best known values without taking implementation details and settings of algorithmic parameters such as maximum number of fitness evaluations into account.

Tables 9–12 show the comparative results of TMSGD-QAP and 9 other well-known metaheuristics for QAPLIB benchmark problems. Since APD and BPD results of DivTS were also available for all problems in this set, scores of TMSGD-QAP and DivTS are organized in two columns. Considering Table 9 that illustrates APD and BPD scores for type-I QAPLIB problems, the hardest problems of this set are *taixxa* and *lipaxxa* instances. Scores for *taixxa* instances are available for all metaheuristics, except GRASP, and TMSGD-QAP performs equally well or better than its competitors in 4 of these 9 QAP instances, whereas ETS1 and GA/TS/I are the two best performing algorithms for the remaining *taixxa* instances. While the

Table 8
Experimental scores with respect to APD, BPD, WPD and CPU time for Drezner's benchmark problems.

Problem	Size	BKS	APD	BPD	WPD	Time (min)
dre15	15	306	$0.0_{(10)}$	0.0	0.0	0.035
dre18	18	332	$0.0_{(10)}$	0.0	0.0	0.124
dre21	21	356	$0.0_{(10)}$	0.0	0.0	0.312
dre24	24	396	$0.0_{(10)}$	0.0	0.0	0.934
dre28	28	476	$0.0_{(10)}$	0.0	0.0	1.314
dre30	30	508	$0.0_{(10)}$	0.0	0.0	2.600
dre42	42	764	$0.250_{(6)}$	0.0	0.377	8.864
dre56	56	1086	$3.556_{(3)}$	1.621	4.733	18.653
dre72	72	1452	8.388	4.105	14.669	47.114
dre90	90	1838	10.979	6.582	15.376	96.551
dre110	110	2264	15.123	9..4966	25.751	161.430
dre132	132	2744	17.553	10.156	31.969	211.122

Table 9
Comparisons with literature for type-I QAPLIB benchmark problems.

Problem	TMSGD-QAP		DivTS		RTS	GRASP	ACO-GA/LS	ETS1	GA/TS	GA/TS/I	I-ILS6	ACO2
	APD	BPD	APD	BPD	APD	BPD	APD	APD	APD	APD	APD	APD
tai20a	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000		0.110	0.000	0.061	0.000		0.191
tai25a	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000		0.290	0.037	0.088	0.000	0.000	0.488
tai30a	$0.091_{(6)}$	0.000	$0.000_{(10)}$	0.000	0.000		0.340	0.003	0.019	0.000	0.000	0.359
tai35a	$0.153_{(2)}$	0.000	$0.000_{(10)}$	0.000	0.112		0.490	0.000	0.126	0.000	0.000	0.773
tai40a	0.261	0.125	$0.222_{(1)}$	0.000	0.462		0.590	0.167	0.338	0.209	0.280	0.933
tai50a	0.276	0.198	0.725	0.607	0.882		0.850	0.322	0.567	0.424	0.610	1.236
tai60a	0.448	0.218	0.718	0.551	0.974		0.820	0.570	0.590	0.547	0.820	1.372
tai80a	0.832	0.289	0.753	0.434	1.065		0.860	0.321	0.271	0.320	0.620	1.134
tai100a	0.874	0.337	0.825	0.703	1.071		0.800	0.367	0.296	0.259	0.690	
rou12	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
rou15	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
rou20	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
lipa20a	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
lipa20b	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
lipa30a	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
lipa30b	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
lipa40a	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
lipa40b	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000				0.000	
lipa50a	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.905	0.000					
lipa50b	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000				0.000	
lipa60a	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.839	0.000					
lipa60b	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000				0.000	
lipa70a	$0.046_{(4)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.746	0.060					
lipa70b	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000				0.000	
lipa80a	0.055	0.042	$0.000_{(10)}$	0.000	0.363	0.676	0.550					
lipa80b	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000				0.000	
lipa90a	0.157	0.049	$0.137_{(7)}$	0.000	0.341	0.615	0.350					
lipa90b	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000				0.000	

proposed algorithm is better than RTS for 5 of 9 *taixxa* problems, it outperforms ACO2 for all problem instances. Considering the performances of TMSGD-QAP and DivTS, that is a successful variant of RTS for QAP, DivTS performs better than TMSGD-QAP in 5 of the 9 problems with respect to the APD scores, however TMSGD-QAP achieved equally well or better scores in 8 of the 9 instances in terms of the BPD scores. DivTs' BPD score is better for *tai40a* problem only. This clearly shows that while exploration capability of the proposed method results in solutions in different regions of the solution space, its intensification capability through using a two-stage memory also makes it possible to locate better solutions even for difficult problem instances. For the second class of difficult problems in type-I instances, *lipaxxa* instances, DivTS performs better than TMSGD-QAP in 3 of 8 problems with respect to APD scores. The proposed method performs equally well-or better than RTS and GRASP in all of 9 problems. It can also be seen in terms of BPD scores that the solutions extracted by TMSGD-QAP for the large size *lipa70a*, *lipa80a*, and *lipa90a* instances are very close to fitness of BKSs. The use of the two-stage archive storing dissimilar solutions with promising fitness values results in spending part of limited fitness evaluations around several solution that delayed the intensification around the most promising solution (best solution found so far). This is main reason of being slightly behind the DivTS algorithm. On the other hand, if take the number of fitness evaluations into account, the number of fitness evaluations used by DivTS is *at least* 50 000*N, whereas this parameter for TMSGD-QAP is *at most* 20 000*N. That is, the proposed method achieved equally well or better solutions for difficult problem instances through 2.5 times less fitness evaluations, except for only a few large-size problems for which APD and BPD values are less than 0.05% in most of the cases. Other type-I problems are simple and they solved optimally for all algorithms for which the corresponding scores are available.

Scores for comparative evaluations of type-II QAPLIB instances are shown in Table 10. The easiest problems of this category are the *nugxx* instances that are solved optimally by all algorithms that attempted to solve them. The hardest problems of type-II instances are the *skoxx* instances and, in terms of the APD scores, I-ILS6 is the best performing algorithm in 7 of the 13 problems. However, GA/TS-IC outperforms all of its competitors for *skoxx* instances of size 90 and above. TMSGD-QAP achieved APD scores of less than

Table 10
Comparisons with literature for Type-II QAPLIB benchmark problems.

Problem	TMSGD-QAP		DivTS		RTS	GRASP	ACO-GA/LS	GA-S/TS	GA/C/TS	GA-IC-TS	I-ILS6	ACO2
	APD	BPD	APD	BPD	APD	BPD	BPD	APD	APD	APD	APD	APD
nug12	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
nug14	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000		0.000					
nug15	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
nug16a	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000		0.000					
nug16b	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000		0.000					
nug17	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000		0.000					
nug18	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000		0.000					
nug20	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
nug21	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000		0.000					
nug22	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000		0.000					
nug24	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000		0.000					
nug25	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000		0.000				0.000	
nug27	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000		0.000				0.000	
nug28	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000		0.000				0.000	
nug30	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000	0.001	0.000		0.000	0.026
sko42	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000		0.000	0.001	0.000		0.000	0.015
sko49	$0.005_{(6)}$	0.000	$0.008_{(7)}$	0.000	0.038		0.060	0.062	0.009		0.000	0.067
sko56	$0.001_{(8)}$	0.000	$0.002_{(8)}$	0.000	0.010		0.010	0.007	0.001	0.000	0.000	0.068
sko64	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.005		0.000	0.019	0.000	0.000	0.000	0.042
sko72	$0.007_{(2)}$	0.000	$0.006_{(2)}$	0.000	0.043		0.020	0.056	0.014	0.000	0.000	0.109
sko81	$0.019_{(1)}$	0.000	$0.016_{(2)}$	0.000	0.051		0.030	0.058	0.014	0.003	0.001	0.071
sko90	$0.031_{(2)}$	0.000	0.026	0.007	0.062		0.040	0.073	0.011	0.001	0.007	0.192
sko100a	$0.029_{(1)}$	0.000	0.027	0.009	0.089		0.020	0.070	0.018	0.002	0.006	
sko100b	$0.015_{(1)}$	0.000	$0.008_{(2)}$	0.000	0.056		0.010	0.042	0.011	0.000	0.012	
sko100c	$0.013_{(1)}$	0.000	$0.006_{(2)}$	0.000	0.031		0.000	0.045	0.003	0.001	0.007	
sko100d	$0.017_{(1)}$	0.000	$0.027_{(1)}$	0.000	0.055		0.030	0.084	0.049	0.000	0.002	
sko100e	$0.016_{(6)}$	0.000	$0.009_{(1)}$	0.000	0.041		0.000	0.028	0.002	0.000	0.021	
sko100f	0.013	0.006	0.023	0.005	0.066		0.030	0.110	0.032	0.003	0.037	
scr12	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
scr15	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
scr20	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
tho30	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000		0.000	0.000	0.000		0.000	
tho40	$0.003_{(8)}$	0.000	$0.001_{(9)}$	0.000	0.011		0.010	0.042	0.010	0.003	0.000	
tho150	0.039	0.013	0.030	0.006	0.078		0.090				0.068	
wil50	$0.002_{(9)}$	0.000	$0.000_{(10)}$	0.000	0.010		0.000	0.011	0.002	0.000	0.000	
wil100	$0.008_{(1)}$	0.000	$0.005_{(1)}$	0.000	0.028		0.010	0.043	0.002	0.000	0.004	

0.03% for all of the *skoxx* instances, however DivTS's APD scores are better for large-size *skoxx* problems, except for *sko49, sko56* and *sko100f* instances. BPD scores of TMSGD-QAP and DivTS are also very close to each other. Among 13 *skoxx* instances, the two algorithms perform equally well in 10 instances; while TMSGD-QAP is better in 2 instances *sko90, sko100a*, DivTS is better in 1 instance *sko100f*. The other four type-II instances for which algorithms' performances differ are *tho40, tho150, wil50*, and *wil100* instances. The best APD score for *tho40* instance belongs to I-ILS6, the second place is taken by DivTS, and the third place is shared by TMSGD-QAP and GA/TS-IC methods. None of the algorithm solved large-size *tho150* optimally, the proposed method is the second best performing algorithm for this problem and both the APD and BPD scores are very close to the best performing DivTS metaheuristic. GA/IC-TS has zero values for APD scores of *wil50* and *wil100* instances while the proposed method and DivTS are performing approximately equally well for these two problems. Based on the results associated with the type-II QAPLIB instances, it can still be observed that the capability of searching around diversely distributed promising solutions has the advantage of locating BKSs for most of the difficult problem instances and approaching BKSs quite closely from different directions, however spending some of limited number of fitness evaluations around multiple promising solutions may cause termination of procedure before exactly locating the optimal solutions.

Type-III QAPLIB instances are the simplest set of problems for the proposed TMSGD-QAP algorithm. As exhibited in Table 11, all problems in this set are solved optimally and the cpu-time required is less than one minute for most of the problem instances. Comparing the proposed algorithm against DivTS, it can be seen that the proposed method is better then than DivTS in three problem instances, *chr20b, chr22b* and *chr25a*, for which the number of times the optimal solution is found over ten trials are all 10 for TMSGD-QAP. Similarly, APD values of the proposed algorithm are better than those of RTS for the three problem instances mentioned above. BPD score of GRASP are comparably poor for these three problems compared to those of TMSGD-QAP and DivTS, while BPD score of ACO-GA/LS is poor for *chr20b* only. Other algorithms reported APD scores for only a few simple problem instances and a comparison with their scores cannot lead strong conclusions.

As explained before, QAP problems in type-IV category are simpler compared to the ones in type-I and type-II categories. Table 12 illustrates the APD and BPD scores for problems in this set. Most of the algorithms solved problems up to size 80 optimally, except GA/TS and ACO2 that have comparably weak scores in 3 of 7 instances of size 60 and below. GA/TS/I seems to be the best performing algorithm in this category. For the larger size instances, *tai80b* and *tai100b*, the proposed method's score is very close to that of DivTS, however the achieved score for tai100b is twice better than that of DivTS. The largest-size *taib150b* is attempted by the proposed method only, for which the APD and BPD values are less than or equal to 0.05%.

Experimental scores associated with the set of *tai75eyy* problem instances are given in Table 13. As explained at the beginning of experimental evaluations, the proposed approach solves all *tai27eyy* and *tai45eyy* instances optimally in all trials. In fact, they are also reported as solved optimally by all algorithms listed in Table 13. Hence, scores associated with these problem instances are not tabulated below. Based on our detailed literature search for metaheuristics used for the solution *taixxeyy* instances, best known results of *tai75eyy* problems and three algorithms attempted to

Table 11
Comparisons with literature for type-III QAPLIB benchmark problems.

Problem	TMSGD-QAP		DivTS		RTS	GRASP	ACO-GA/LS	GA-C/TS	GA/TS	GA/TS/I	I-ILS6	ACO2
	APD	BPD	APD	BPD	APD	BPD	BPD	APD	APD	APD	APD	APD
bur26a-h	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000				0.000	0.006
chr12a	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
chr12b	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
chr12c	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
chr15a	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
chr15b	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
chr15c	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
chr18a	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
chr1bb	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
chr20a	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
chr20b	$0.000_{(10)}$	0.000	$0.200_{(9)}$	0.000	0.400	3.133	0.870					
chr20c	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000					
chr22a	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.552	0.000					
chr22b	$0.000_{(10)}$	0.000	$0.152_{(8)}$	0.000	0.213	1.421	0.000					
chr25a	$0.000_{(10)}$	0.000	$0.943_{(5)}$	0.000	0.601	4.636	0.000		0.232	0.000	0.000	
els19	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000		0.000	0.000		
had12-20	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000		0.000					
kra30a	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000		0.224	0.000	0.000	
kra30b	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.00	0.000	0.000	0.028	0.000	0.000	
kra32	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000			0.000			0.000	
ste36a	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.651	0.000		0.061	0.000	0.000	
ste36b	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000	0.005	0.000	0.000	0.000	
ste36c	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.188	0.000	0.000	0.001	0.000	0.000	
esc16a	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000	0.039				
esc16b	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000					
esc16c	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000					
esc16d	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000					
esc16e	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000					
esc16f	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000					
esc16g	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000					0.134
esc16h	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000					0.023
esc16i	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000					
esc16j	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000					0.036
esc32a	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000		0.154	0.000		0.000
esc32b	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000	0.000				
esc32c	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000	0.000			0.000	
esc32d	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000	0.000			0.000	
esc32e	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000	0.000			0.000	
esc32f	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000				0.000	
esc32g	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000				0.000	
esc32h	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000				0.000	
esc64a	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	
esc128	$0.000_{(10)}$	0.000	$0.0_{(10)}$	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	

Table 12
Comparisons with literature for type-IV QAPLIB benchmark problems.

Problem	TMSGD-QAP		DivTS		RTS	ACO-GA/LS	GA/ETS1	GA/TS	TS/I	I-ILS6	ACO2	ACO3
	APD	BPD	APD	BPD	APD	BPD	APD	APD	APD	APD	APD	APD
tai20b	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
tai25b	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000	0.007	0.000	0.000	0.000	0.000
tai30b	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
tai35b	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000	0.059	0.000	0.000	0.051	0.000
tai40b	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.402	0.000
tai50b	$0.000_{(10)}$	0.000	$0.000_{(10)}$	0.000	0.000	0.000	0.000	0.002	0.000	0.033	0.172	0.002
tai60b	$0.005_{(5)}$	0.000	$0.000_{(8)}$	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.005	0.005
tai64c	$0.000_{(10)}$	0.000										
tai80b	0.007	0.004	0.006	0.001	0.008	0.000	0.008	0.003	0.000	0.383	0.591	0.096
tai100b	0.028	0.011	0.056	0.005	0.008	0.010	0.072	0.014	0.000	0.083		
tai150b	0.051	0.023										

solve them were available. References from which the corresponding scores are taken, with the names specified in their publications, are also shown in Table 13. Then, since those three metaheuristics' published scores were based on 20 run, each of these problems is solved over 20 independent runs for comparative evaluations and the corresponding APD values are listed in Table 13. It is clear that the best performing algorithm is HG. The proposed method takes the second place and performs significantly better than cooperative parallel tabu search (CPTS) algorithm. GH, for which the scores of *tai75e01* is available only, has a poor performance compared to the other algorithms under consideration. GH is also used for the solution *tai125e01* for which the score of TMSGD-QAP is twice better. These results demonstrate that the proposed approach is capable of extracting optimal solution for large-size difficult QAP instances while its average performance is also within acceptable limits.

Table 14 illustrates the APD scores for the final set of QAP instances containing the difficult *drexx* problems. Unfortunately, only one algorithm is found in literature with scores of some of

Table 13
Comparisons with literature for Taixxeyy benchmark problems.

Problem	TMSGD-QAP APD	HG [70] APD	CPTS [68] APD	GH [21] APD
Tai75e01	$4.097_{(8)}$	$0.175_{(14)}$	5.915	10.100
Tai75e02	$4.165_{(8)}$	$0.339_{(14)}$	8.751	
Tai75e03	$1.081_{(14)}$	$0.000_{(20)}$	2.681	
Tai75e04	$1.005_{(16)}$	$0.030_{(19)}$	3.705	
Tai75e05	$1.412_{(11)}$	$0.000_{(20)}$	2.412	
Tai75e06	$2.115_{(10)}$	$0.000_{(20)}$	4.005	
Tai75e07	$4.662_{(7)}$	$0.136_{(17)}$	8.015	
Tai75e08	$3.951_{(8)}$	$0.059_{(19)}$	8.523	
Tai75e09	$1.051_{(15)}$	$0.000_{(20)}$	4.798	
Tai75e10	$2.910_{(10)}$	$0.149_{(19)}$	5.740	
Tai75e11	$1.021_{(16)}$	$0.000_{(20)}$	3.721	
Tai75e12	$4.619_{(7)}$	$0.000_{(20)}$	8.196	
Tai75e13	$2.922_{(11)}$	$0.000_{(20)}$	5.123	
Tai75e14	$1.601_{(12)}$	$0.000_{(20)}$	3.659	
Tai75e15	$1.007_{(16)}$	$0.000_{(20)}$	2.976	
Tai75e16	$1.081_{(14)}$	$0.055_{(19)}$	3.389	
Tai75e17	$1.223_{(12)}$	$0.129_{(17)}$	3.653	
Tai75e18	$3.033_{(9)}$	$0.000_{(20)}$	7.176	
Tai75e19	$1.044_{(15)}$	$0.000_{(20)}$	2.861	
Tai75e20	$1.374_{(14)}$	$0.053_{(19)}$	2.743	
tai125e01	7.751			15.600

Table 14
Comparisons with literature for Drexx benchmark problems.

Problem	TMSGD-QAP APD	HG [70] APD
dre15	$0.000_{(10)}$	
dre18	$0.000_{(10)}$	
dre21	$0.000_{(10)}$	
dre24	$0.000_{(10)}$	
dre28	$0.000_{(10)}$	
dre30	$0.000_{(10)}$	$0.000_{(20)}$
dre42	$0.250_{(6)}$	$1.340_{(18)}$
dre56	$3.566_{(3)}$	$17.460_{(6)}$
dre72	8.388	$27.280_{(2)}$
dre90	10.979	33.88
dre110	15.123	
dre132	17.553	

the *drexx* problems. It is observed that the proposed TMSGD-QAP algorithm solved all *drexx* problem of size 30 and below optimally in all 10 trials. It was also capable of extracting optimal solutions of *dre42* and *dre56* instances. For rest of the problems in this set, TMSGD-QAP's APD score increases with increasing problem size, however its performance compared to HG, that was the best performing algorithm for *tai75eyy* instances, is significantly better. Remembering that the basic characteristic of *drexx* problems is that many solutions within the neighbourhood optimal solutions have the same fitness values, the use of a two-stage external memory with promising solutions from distant regions of the solution space made it possible to reach better solutions compared to algorithms that concentrate the search around a few initially promising solutions.

6.3. Comparison with latest algorithms

Three very recently published methods for the solution of QAP are taken into consideration for further comparative evaluations of the proposed algorithm TMSGD-QAP. The first of these methods, proposed by Tosun [71], is a study on the performance of parallel hybrid algorithms (PHA) for the solution of QAP. This hybrid approach combined island parallel genetic algorithms (IPGA) with robust tabu search algorithm (RTS). It is implemented on a cluster computer with 92 cpus, a total of 368 cores, 736 GB of total memory and 6.5 TB of permanent storage capacity. This rich computational resources made it possible to use huge population sizes (5000 individuals for each slave) and a termination condition of approximately 2 million number of failures for the RTS algorithm. As indicated in Table 1, the proposed TSMGD-QAP employs RTS over a few solutions with a total number fitness evaluations equal to $10*N$ and with much less frequency compared to PHA. Results of PHA and TSMGD-QAP for four types of QAPLIB benchmarks are listed in tables below together with comparative evaluations and interpretations.

The second algorithm authored by Benlic et al. is a memetic search algorithm that integrates breakout local search (BLS) within an evolutionary computation framework [72]. This algorithm is executed on a single processor with a maximum of time limit of 2 hours. The authors used BLS for each newly generated offspring after crossover with 5000 local search iterations. This algorithm is named as BMA and its associated results are presented in the following tables.

The third algorithm under consideration is proposed by Dokeroglu and it is a hybrid teaching-learning based optimization approach for QAP. The algorithm first optimizes a population of individuals using a teaching-learning based optimization (TLBO) procedure [73]. Individuals within the latest population optimized by TLBO are further improved by the RTS algorithm. Maximum number of failures for RTS is taken as $2000*N$. The authors named this algorithm as TLBO-RTS.

Tables 15–18 illustrate the results of TMSGD-QAP, PHA, BMA and TLBO-RTS for the four types of QAPLIB benchmark problems. As explained above, a fair comparison of all algorithms is not possible due to variations in computational resources (both in hardware and software), number of fitness evaluations, termination conditions, and algorithm parameters such as population size, crossover rate, local search parameters etc. Hence, the cpu-time columns in the following tables are taken from the corresponding references as they are and will be used for performance interpretations rather than comparing algorithm complexities. Table 15 shows results of four algorithms for type-I benchmark problems that contain hard problems for metaheuristics. BMA seems to be the best performing algorithm in terms of solution quality, computational resources and cpu time requirements. Noting that PHA is running on 46 nodes with a population of 5000 individuals on each node (i.e, total population size is 46*5000) and comparably very large local search iterations, its performance is not proportional to richness of its computational resources. For example BMA is better than PHA even though it runs on a single cpu and with a population size of 15 individuals. In this respect, considering that TMSGD-QAP is a trajectory based algorithm (i.e. population size is 1), running on a single cpu and using RTS with much less frequency over a few randomly selected individuals, its performance can certainly be considered better than that of PHA. Problems for which PHA seems to be better than TMSGD-QAP, corresponding cpu times of PHA are 3–5 times larger to achieve around 20% improvement in solution quality. Considering also the number of cpus and storage space used to get this improvement, it can obviously be claimed that it is not worth of this trade off. It can also be seen from Table 15 that TMSGD-QAP is better than TLBO-RTS in terms of both solution quality and cpu-times.

For type-II benchmark problems, all algorithms solved *nugxx* problems optimally however PHA took 10 to 100 times more cpu-time compared to TMSGD-QAP. The fastest algorithm for this set of problems is BMA. For *skoxx* problems, BMA is again the best performing algorithm whereas TLBO-RTS takes the fourth position in terms of solution quality. PHA solved all *skoxx* instances optimally in 3 to 10 times more cpu-time than TMSGD-QAP whereas the solution quality improvement stayed in the order 0.001–0.003%. Success of TMSGD-QAP is still better than that of TLBO-RTS for this set of problems also. Type-II benchmarks contains a large size instance *tho150* for which TMSGD-QAP performed better than BMA

Table 15
Comparisons with recently published algorithms for type-I QAPLIB benchmark problems.

Problem	TMSGD-QAP		PHA		BMA		TLBO-RTS	
	APD	Time (min)	APD	Time (min)	APD	Time (min)	APD	Time (min)
tai20a	$0.000_{(10)}$	2.096	$0.000_{(10)}$	3.73	$0.000_{(100)}$	0.000	$0.000_{(10)}$	0.3
tai25a	$0.000_{(10)}$	15.820	$0.000_{(10)}$	5.50	$0.000_{(100)}$	0.000	$0.000_{(10)}$	0.3
tai30a	$0.091_{(6)}$	20.291	$0.000_{(10)}$	8.89	$0.000_{(100)}$	0.000	$0.000_{(10)}$	1.00
tai35a	$0.153_{(2)}$	24.991	$0.000_{(10)}$	12.34	$0.000_{(100)}$	0.000	$0.150_{(5)}$	1.50
tai40a	0.261	27.783	$0.000_{(10)}$	100.83	$0.059_{(2)}$	8.1	0.440	2.30
tai50a	0.276	41.139	$0.000_{(10)}$	127.4	$0.131_{(2)}$	42	0.961	4.4
tai60a	0.448	78.862	$0.000_{(10)}$	195.8	$0.144_{(2)}$	67.5	1.098	7.6
tai80a	0.832	111.337	0.644	399.7	0.426	65.8	1.124	18.8
tai100a	0.874	138.321	0.537	718.9	0.405	44.1	1.091	36.1
rou12	$0.000_{(10)}$	0.022	$0.000_{(10)}$	1.46	$0.000_{(100)}$	0.003	$0.000_{(10)}$	0.10
rou15	$0.000_{(10)}$	0.044	$0.000_{(10)}$	2.23	$0.000_{(100)}$	0.007	$0.000_{(10)}$	0.10
rou20	$0.000_{(10)}$	1.716	–	–	$0.000_{(100)}$	0.008	$0.000_{(10)}$	0.30
lipa20a	$0.000_{(10)}$	0.044	$0.000_{(10)}$	3.71	$0.000_{(100)}$	0.002	$0.000_{(10)}$	0.30
lipa20b	$0.000_{(10)}$	0.033	–	–	$0.000_{(100)}$	0.000	$0.000_{(10)}$	0.30
lipa30a	$0.000_{(10)}$	0.125	$0.000_{(10)}$	9.09	$0.000_{(100)}$	0.003	$0.000_{(10)}$	1.00
lipa30b	$0.000_{(10)}$	0.057	$0.000_{(10)}$	8.93	$0.000_{(100)}$	0.002	$0.000_{(10)}$	1.00
lipa40a	$0.000_{(10)}$	0.682	$0.000_{(10)}$	15.21	$0.000_{(100)}$	0.018	$0.000_{(10)}$	2.30
lipa40b	$0.000_{(10)}$	0.107	–	–	$0.000_{(100)}$	0.005	$0.000_{(10)}$	2.30
lipa50a	$0.000_{(10)}$	3.901	$0.000_{(10)}$	126.37	$0.000_{(100)}$	0.025	$0.000_{(10)}$	4.30
lipa50b	$0.000_{(10)}$	4.092	–	–	$0.000_{(100)}$	0.007	$0.000_{(10)}$	4.30
lipa60a	$0.000_{(10)}$	22.064	$0.000_{(10)}$	195.06	$0.000_{(100)}$	0.137	$0.000_{(10)}$	7.60
lipa60b	$0.000_{(10)}$	1.909	–	–	$0.000_{(100)}$	0.007	$0.000_{(10)}$	7.60
lipa70a	$0.046_{(4)}$	34.877	$0.000_{(10)}$	298.78	$0.000_{(100)}$	0.512	$0.000_{(10)}$	12.5
lipa70b	$0.000_{(10)}$	1.759	–	–	$0.000_{(100)}$	0.025	$0.000_{(10)}$	12.5
lipa80a	0.055	49.877	$0.000_{(10)}$	394.41	$0.000_{(100)}$	0.853	$0.254_{(5)}$	18.8
lipa80b	$0.000_{(10)}$	3.700	–	–	$0.000_{(100)}$	0.068	$0.000_{(10)}$	18.8
lipa90a	0.157	91.442	$0.000_{(10)}$	407.48	$0.000_{(100)}$	2,485	$0.193_{(6)}$	25.7
lipa90b	$0.000_{(10)}$	9.774	–	–	$0.000_{(100)}$	0.069	$0.000_{(10)}$	25.7

Table 16
Comparisons with recently published algorithms for type-II QAPLIB benchmark problems.

Problem	TMSGD-QAP		PHA		BMA		TLBO-RTS	
	APD	Time (min)	APD	Time (min)	APD	Time (min)	APD	Time (min)
nug12	$0.000_{(10)}$	0.013	$0.000_{(10)}$	1.48	$0.000_{(100)}$	0.002	$0.000_{(10)}$	0.10
nug14	$0.000_{(10)}$	0.021	$0.000_{(10)}$	1.92	$0.000_{(100)}$	0.002	$0.000_{(10)}$	0.10
nug15	$0.000_{(10)}$	0.033	$0.000_{(10)}$	2.22	$0.000_{(100)}$	0.003	$0.000_{(10)}$	0.10
nug16a	$0.000_{(10)}$	0.025	$0.000_{(10)}$	2.49	$0.000_{(100)}$	0.003	$0.000_{(10)}$	0.20
nug16b	$0.000_{(10)}$	0.029	–	–	$0.000_{(100)}$	0.005	$0.000_{(10)}$	0.20
nug17	$0.000_{(10)}$	0.040	$0.000_{(10)}$	2.78	$0.000_{(100)}$	0.005	$0.000_{(10)}$	0.20
nug18	$0.000_{(10)}$	0.058	$0.000_{(10)}$	3.07	$0.000_{(100)}$	0.005	$0.000_{(10)}$	0.20
nug20	$0.000_{(10)}$	0.068	$0.000_{(10)}$	3.73	$0.000_{(100)}$	0.007	$0.000_{(10)}$	0.30
nug21	$0.000_{(10)}$	0.091	–	–	$0.000_{(100)}$	0.007	$0.000_{(10)}$	0.30
nug22	$0.000_{(10)}$	0.038	$0.000_{(10)}$	4.46	$0.000_{(100)}$	0.007	$0.000_{(10)}$	0.40
nug24	$0.000_{(10)}$	0.103	$0.000_{(10)}$	5.23	$0.000_{(100)}$	0.011	$0.000_{(10)}$	0.50
nug25	$0.000_{(10)}$	0.088	$0.000_{(10)}$	6.49	$0.000_{(100)}$	0.011	$0.000_{(10)}$	0.50
nug27	$0.000_{(10)}$	0.129	$0.000_{(10)}$	7.52	$0.000_{(100)}$	0.008	$0.000_{(10)}$	0.70
nug28	$0.000_{(10)}$	0.382	–	–	$0.000_{(100)}$	0.012	$0.000_{(10)}$	0.80
nug30	$0.000_{(10)}$	1.783	$0.000_{(10)}$	9.01	$0.000_{(100)}$	0.030	$0.000_{(10)}$	0.90
sko42	$0.000_{(10)}$	2.242	$0.000_{(10)}$	16.54	$0.000_{(100)}$	0.022	$0.000_{(10)}$	2.40
sko49	$0.005_{(6)}$	3.849	$0.000_{(10)}$	40.18	$0.000_{(100)}$	0.008	$0.037_{(1)}$	4.00
sko56	$0.001_{(8)}$	14.721	$0.000_{(10)}$	162.4	$0.000_{(100)}$	0.017	$0.005_{(6)}$	6.00
sko64	$0.000_{(10)}$	29.385	$0.000_{(10)}$	230.78	$0.000_{(100)}$	0.025	$0.002_{(6)}$	9.20
sko72	$0.007_{(2)}$	37.992	$0.000_{(10)}$	336.3	$0.000_{(10)}$	3.5	0.024	13.2
sko81	$0.019_{(1)}$	57.141	$0.000_{(10)}$	398.7	$0.000_{(10)}$	4.3	–	–
sko90	$0.031_{(2)}$	93.826	$0.000_{(10)}$	405.3	$0.000_{(10)}$	15.3	0.049	25.20
sko100a	$0.029_{(1)}$	153.173	$0.000_{(10)}$	420.93	$0.000_{(10)}$	22.3	0.064	35.70
sko100b	$0.015_{(1)}$	164.272	$0.000_{(10)}$	423.2	$0.000_{(10)}$	6.5	0.017	35.70
sko100c	$0.013_{(1)}$	154.514	$0.000_{(10)}$	410.63	$0.000_{(10)}$	12.0	0.019	35.70
sko100d	$0.017_{(1)}$	148.855	$0.000_{(10)}$	419.1	$0.006_{(9)}$	20.9	0.031	35.70
sko100e	$0.016_{(6)}$	146.145	$0.000_{(10)}$	423.05	$0.000_{(10)}$	11.9	0.023	35.70
sko100f	0.013	153.383	$0.000_{(10)}$	419.8	$0.000_{(10)}$	23.0	0.047	35.70
scr12	$0.000_{(10)}$	0.013	–	–	$0.000_{(100)}$	0.002	$0.000_{(10)}$	0.10
scr15	$0.000_{(10)}$	0.047	–	–	$0.000_{(100)}$	0.005	$0.000_{(10)}$	0.10
scr20	$0.000_{(10)}$	0.138	–	–	$0.000_{(100)}$	0.008	$0.000_{(10)}$	0.30
tho30	$0.000_{(10)}$	5.034	$0.000_{(10)}$	8.84	$0.000_{(100)}$	0.005	$0.000_{(10)}$	1.00
tho40	$0.003_{(8)}$	14.024	–	–	$0.000_{(100)}$	0.007	$0.000_{(10)}$	2.20
tho150	0.039	512.787	0.009	1767.99	$0.080_{(3)}$	416.4	0.060	119.00
wil50	$0.002_{(9)}$	18.811	$0.000_{(10)}$	126.3	$0.000_{(100)}$	0.393	$0.000_{(10)}$	4.30
wil100	$0.008_{(1)}$	155.128	$0.000_{(10)}$	–	$0.000_{(10)}$	14.5	0.010	36.10

Table 17
Comparisons with recently published algorithms for type-III QAPLIB benchmark problems.

Problem	TMSGD-QAP		PHA		BMA		TLBO-RTS	
	APD	Time (min)	APD	Time (min)	APD	Time (min)	APD	Time (min)
bur26a-h	$0.000_{(10)}$	0.249	$0.000_{(10)}$	7.56	$0.000_{(100)}$	0.007	$0.000_{(10)}$	0.60
chr12a	$0.000_{(10)}$	0.155	$0.000_{(10)}$	1.47	$0.000_{(100)}$	0.003	$0.000_{(10)}$	0.10
chr12b	$0.000_{(10)}$	0.080	$0.000_{(10)}$	1.55	$0.000_{(100)}$	0.003	$0.000_{(10)}$	0.10
chr12c	$0.000_{(10)}$	0.185	$0.000_{(10)}$	1.47	$0.000_{(100)}$	0.003	$0.000_{(10)}$	0.10
chr15a	$0.000_{(10)}$	0.261	$0.000_{(10)}$	2.23	$0.000_{(100)}$	0.005	$0.000_{(10)}$	0.10
chr15b	$0.000_{(10)}$	0.249	–	–	$0.000_{(100)}$	0.007	$0.000_{(10)}$	0.10
chr15c	$0.000_{(10)}$	0.269	–	–	$0.000_{(100)}$	0.005	$0.000_{(10)}$	0.10
chr18a	$0.000_{(10)}$	0.352	–	–	$0.000_{(100)}$	0.007	$0.000_{(10)}$	0.20
chr18b	$0.000_{(10)}$	0.354	–	–	$0.000_{(100)}$	0.008	$0.000_{(10)}$	0.20
chr20a	$0.000_{(10)}$	0.649	$0.000_{(10)}$	3.73	$0.000_{(100)}$	0.035	$0.000_{(10)}$	0.30
chr20b	$0.000_{(10)}$	0.840	–	–	$0.000_{(100)}$	0.095	$0.000_{(10)}$	0.30
chr20c	$0.000_{(10)}$	0.588	–	–	$0.000_{(100)}$	0.013	$0.000_{(10)}$	0.30
chr22a	$0.000_{(10)}$	0.853	–	–	$0.000_{(100)}$	0.027	$0.000_{(10)}$	0.40
chr22b	$0.000_{(10)}$	0.897	–	–	$0.000_{(100)}$	0.030	$0.000_{(10)}$	0.40
chr25a	$0.000_{(10)}$	0.972	–	–	$0.000_{(100)}$	0.158	$0.000_{(10)}$	0.50
els19	$0.000_{(10)}$	0.088	–	–	$0.000_{(100)}$	0.0.007	$0.000_{(10)}$	0.20
had12–20	$0.000_{(10)}$	0.010	$0.0_{(10)}$	3.07	$0.000_{(100)}$	0.004	$0.000_{(10)}$	0.16
kra30a	$0.000_{(10)}$	0.947	–	–	$0.000_{(100)}$	0.080	$0.000_{(10)}$	1.00
kra30b	$0.000_{(10)}$	1.392	–	–	$0.000_{(100)}$	0.020	$0.000_{(10)}$	1.00
kra32	$0.000_{(10)}$	0.626	–	–	$0.000_{(100)}$	0.018	$0.000_{(10)}$	1.20
ste36a	$0.000_{(10)}$	2.173	$0.0_{(10)}$	12.76	$0.000_{(100)}$	0.082	$0.000_{(10)}$	1.60
ste36b	$0.000_{(10)}$	0.406	–	–	$0.000_{(100)}$	0.022	$0.000_{(10)}$	1.60
ste36c	$0.000_{(10)}$	1.395	$0.0_{(10)}$	–	$0.000_{(100)}$	0.087	$0.000_{(10)}$	1.60
esc16a	$0.000_{(10)}$	0.003	$0.0_{(10)}$	2.53	$0.000_{(100)}$	0.000	$0.000_{(10)}$	0.10
esc16b	$0.000_{(10)}$	0.001	–	–	$0.000_{(100)}$	0.000	$0.000_{(10)}$	0.10
esc16c	$0.000_{(10)}$	0.004	–	–	$0.000_{(100)}$	0.000	$0.000_{(10)}$	0.10
esc16d	$0.000_{(10)}$	0.001	–	–	$0.000_{(100)}$	0.000	$0.000_{(10)}$	0.10
esc16e	$0.000_{(10)}$	0.002	–	–	$0.000_{(100)}$	0.000	$0.000_{(10)}$	0.10
esc16f	$0.000_{(10)}$	0.046	–	–	$0.000_{(100)}$	0.002	$0.000_{(10)}$	0.10
esc16g	$0.000_{(10)}$	0.002	–	–	$0.000_{(100)}$	0.002	$0.000_{(10)}$	0.10
esc16h	$0.000_{(10)}$	0.001	$0.0_{(10)}$	2.47	$0.000_{(100)}$	0.000	$0.000_{(10)}$	0.10
esc16i	$0.000_{(10)}$	0.001		–	$0.000_{(100)}$	0.000	$0.000_{(10)}$	0.10
esc16j	$0.000_{(10)}$	0.001	–	–	$0.000_{(100)}$	0.000	$0.000_{(10)}$	0.10
esc32a	$0.000_{(10)}$	2.209	–	–	$0.000_{(100)}$	0.000	$0.000_{(10)}$	1.20
esc32b	$0.000_{(10)}$	0.598	–	–	$0.000_{(100)}$	0.002	$0.000_{(10)}$	1.20
esc32c	$0.000_{(10)}$	0.005	–	–	$0.000_{(100)}$	0.002	$0.000_{(10)}$	1.20
esc32d	$0.000_{(10)}$	0.026	–	–	$0.000_{(100)}$	0.002	$0.000_{(10)}$	1.20
esc32e	$0.000_{(10)}$	0.004	–	–	$0.000_{(100)}$	0.002	$0.000_{(10)}$	1.20
esc32f	$0.000_{(10)}$	0.004	–	–	–	–	–	–
esc32g	$0.000_{(10)}$	0.005	–	–	$0.000_{(100)}$	0.000	$0.000_{(10)}$	1.20
esc32h	$0.000_{(10)}$	0.005	–	–	$0.000_{(100)}$	0.002	$0.000_{(10)}$	1.20
esc64a	$0.000_{(10)}$	0.038	–	–	$0.000_{(100)}$	0.003	$0.000_{(10)}$	9.20
esc128	$0.000_{(10)}$	2.135	–	–	$0.000_{(100)}$	0.022	$0.000_{(10)}$	75.00

and TLBO-RTS algorithms in terms of the solution quality. PHA's solution quality is the best for this particular instance while its running time is 3 to 10 times more than those of other methods.

Type-III problems are the simplest ones that are solved optimally by the four algorithms. PHA is used to solve a few of these problems for which it is the slowest algorithm. The fastest algorithm for this set of problems is BMA whereas the speed of TMSGD-QAP is faster than that of TLBO-RTS for majority of instances.

For majority of problems in type-IV category of QAPLIB problems, TMSGD-QAP and its competitors exhibited optimal performance. Similar to the previous comments, PHA is still the slowest and BMA is the fastest and the most successful algorithm. For a few problems in this category, success of TMSGD-QAP and TLBO-RTS are similar to each other. For the second largest QAPLIB problem *tai150b*, TMSGD-QAP performed slightly better than BMA, PHA is the best performing algorithm that runs almost 7 times slower than TMSGD-QAP.

Table 18
Comparisons with recently published algorithms for type-IV QAPLIB benchmark problems.

Problem	TMSGD-QAP		PHA		BMA		TLBO-RTS	
	APD	Time (min)	APD	Time (min)	APD	Time (min)	APD	Time (min)
tai20b	$0.000_{(10)}$	0.074	$0.000_{(10)}$	3.70	$0.000_{(100)}$	0.000	$0.000_{(10)}$	0.26
tai25b	$0.000_{(10)}$	0.327	$0.000_{(10)}$	5.70	$0.000_{(100)}$	0.000	$0.000_{(10)}$	0.55
tai30b	$0.000_{(10)}$	1.148	$0.000_{(10)}$	8.10	$0.000_{(100)}$	0.000	$0.000_{(10)}$	1.0
tai35b	$0.000_{(10)}$	6.387	$0.000_{(10)}$	11.10	$0.000_{(100)}$	0.000	$0.000_{(10)}$	1.40
tai40b	$0.000_{(10)}$	4.877	$0.000_{(10)}$	15.70	$0.000_{(100)}$	0.000	$0.000_{(10)}$	2.20
tai50b	$0.000_{(10)}$	10.249	$0.000_{(10)}$	58.20	$0.000_{(10)}$	1.2	$0.000_{(10)}$	4.20
tai60b	$0.005_{(5)}$	33.639	$0.000_{(10)}$	94.9	$0.000_{(10)}$	5.2	$0.000_{(10)}$	7.20
tai64c	$0.000_{(10)}$	0.005	$0.000_{(10)}$	–	$0.000_{(100)}$	0.040	–	–
tai80b	0.007	72.601	$0.000_{(10)}$	277.0	$0.000_{(10)}$	31.3	0.005	17.50
tai100b	0.028	138.533	$0.000_{(10)}$	425.0	$0.000_{(10)}$	13.6	$0.027_{(2)}$	35.70
tai150b	0.051	257.967	0.026	1777.48	$0.060_{(1)}$	78.1	–	–

As an overall summary of the above discussions on the performance of TMSGD-QAP against three recently published algorithms, it can be concluded that the proposed two-stage memory support to the trajectory based GDA resulted in a comparably powerful algorithm against its three competitors. BMA is clearly better than our proposal whereas TMSGA-QAP is clearly better than TLBO-RTS and comparisons of these three algorithms are easier and fair since they are executed on similar computing platforms. On the other hand, APH uses enormously larger population size, computing processors, storage space and local search iterations, but it performs worse than BMA that runs on a single cpu with a population of 15 elements. Similarly, APH's performance compared to our trajectory based resource-efficient proposal is not proportional to its huge computational resources. As explained above, APH conducts its search over 46*5000 individuals and takes 3–10 times more cputime for an improvement of approximately 17%. These comparative evaluations are all positive evidences on the effectiveness of the proposed two-stage memory support for metaheuristics.

7. Conclusions and future work

This study aims to exhibit the effectiveness of the novel two-stage external memory architecture, implemented within the framework of a trajectory based metaheuristic, for the solution of a difficult combinatorial optimization problem, the quadratic assignment problem. The fundamental characteristics of the two-stage memory are maintaining a short-term memory, M_{FS}, based of fitness similarity to catch ongoing improvements in the search space and enabling search around multiple promising (by fitness similarity) and diversely distributed (by positional dissimilarity) solutions thorough using the elements in the long-term second-stage memory M_{SS}.

The results obtained for three different sets of QAP benchmarks demonstrated that, the proposed method maintaining a two-stage external memory is a competitive and effective algorithm that was capable of locating optimal solutions for most of the benchmark problem instances. In addition to this, the capability of searching over multiple diversely distributed solutions resulted in extraction of solutions of very-near optimal quality for large-size difficult problem instances. The method was also significantly better in performing search over complex neighbourhoods such as the case in *drexx* problems. Comparisons with the recently published algorithms also exhibit effectiveness of the proposed method against majority of its competitors.

Based on the success achieved over a trajectory based metaheuristic, it can be concluded that the proposed two-stage memory architecture improved the search capability of the underlying metaheuristic framework. Further research is ongoing to implement the proposed method for population-based metaheuristics and for multiobjective optimization combinatorial optimization problems.

References

[1] G. Dueck, New optimization heuristics: the great deluge algorithm and the record-to-record travel, J. Comput. Phys. 104 (1993) 86–92.
[2] E. Burke, Y. Bykov, J. Newall, S. Petrovic, A Time-Predefined Local Search Approach to Exam Timetabling Problems, CRC Press, 2003, pp. 76–90.
[3] E. Burke, Y. Bykov, Solving exam timetabling problems with the flex-deluge algorithm, in: Proceedings of the Practice and Theory of Automated Timetabling Conference (PATAT2006), 2006, pp. 370–372.
[4] V. Ravi, Modified great deluge algorithm versus other metaheuristics in reliability optimization, Asia-Pac. J. Oper. Res. 21 (4) (2004) 487–497.
[5] D. Silva, J.H. Orbit, Great deluge with non-linear decay rate for solving course timetabling problems, in: International IEEE Conference on Intelligent Systems, 2008, pp. 11–18.
[6] P. McMullan, An extended implementation of the great deluge algorithm for course timetabling, in: International Conference on Computational Science – ICCS2007, 2007, pp. 538–545.
[7] N. Nahas, M. Nourelfath, D.A. -Kadi, Iterated Great Deluge for the Dynamic Facility Layout Problem, Technical Report CIRRELT-2010-20, University of Montreal, 2010.
[8] E. Ozcan, M. Misir, G. Ochoa, E.K. Burke, A reinforcement learning-great deluge algorithm hyper-heuristic for examination timetabling, J. Metaheuristic Comput. 1 (1) (2010) 39–59.
[9] N.R. Al-Milli, Hybrid genetic algorithms with great deluge for course timetabling, Int. J. Comput. Sci. Netw. Secur. 10 (4) (2010) 283–288.
[10] D. Landa-Silva, J.H. Obit, Evolutionary non-linear great deluge for university course timetabling, in: International Conf. on Hybrid Intelligent Systems, 2009, pp. 269–276.
[11] A. Abuhamdah, M. Ayob, Hybridization multi-neighbourhood particle collision algorithm and great deluge for solving course timetabling problems, in: Conference on Data Mining and Optimization, 2009, pp. 108–114.
[12] S. Abdullah, K. Shaker, B. McCollum, P. McMullan, Construction of course timetables based on great deluge and tabu search, in: VIII Metaheuristic International Conference, 2009, pp. x1–x12.
[13] S. Abdullah, H. Turabieh, B. McCollum, A hybridization of electromagnetic-like mechanism and great deluge for examination timetabling problems, in: 6th International Workshop on Hybrid Metaheuristics, 2009, pp. 60–72.
[14] S. Ghatei, F.T. Panahi, M. Hosseinzadeh, M. Rouhi, I. Rezazadeh, A. Naebi, Z. Ghatei, R.P. Khajei, A new hybrid algorithm for optimization using PSO and GDA, J. Basic Appl. Sci. Res. 2 (3) (2012) 2336–2341.
[15] A. Acan, An external memory implementation in ant colony optimization, in: Lecture Notes in Computer Science, LNC, 3172, 2004, pp. 73–82.
[16] A. Acan, An external partial permutations memory for ant colony optimization, in: Proceedings of Evolutionary Computation in Combinatorial Optimization – EvoCOP 2005, LNCS 3448, Springer, 2005, pp. 1–11.
[17] J. Montgomery, M. Randall, The accumulated experience ant colony for the travelling salesman problem, Int. J. Comput. Intell. Appl. World Sci. Publ. Co. 3 (2) (2003) 189–198.
[18] T. Koopmans, M. Beckmann, Assignment problems and the location of economic activities, Econometrica 25 (1957) 53–76.
[19] T. Stützle, S. Fernandes, New benchmark instances for the QAP and the experimental analysis of algorithms, in: J. Gottlieb, G.R. Raidl (Eds.), Evolutionary Computation in Combinatorial Optimization – EvoCOP 2004, 2004, pp. 199–209.
[20] M.M. Drugan, Generating QAP instances with known optimum solution and additively decomposable cost function, J. Comb. Optim. (2013), http://dx.doi.org/10.1007/s10878-013-9689-6
[21] Z. Drezner, P.M. Hahn, E.D. Taillard, Recent advances for the quadratic assignment problem with special emphasis on instances that are difficult for metaheuristic methods, Ann. Oper. Res. 139 (2005) 65–94.
[22] E.M. Loiola, N.M.M. de Abreu, P.O.B. Netto, P. Hahn, T. Querido, A survey of the quadratic assignment problem, Eur. J. Oper. Res. 176 (2007) 657–690.
[23] C.A. Floudas, P.M. Pardalos (Eds.), Encyclopedia of Optimization, Springer, 2009 (3119-31-49).
[24] G. Paul, Comparative performance of tabu search and simulated annealing heuristics for the quadratic assignment problem, Oper. Res. Lett. 38 (2010) 577–581.
[25] M.S. Hussin, T. Stu"ztle, Tabu search vs. simulated annealing as a function of the size of quadratic assignment problem instances, Comput. Oper. Res. 43 (2014) 286–291.
[26] E. Taillard, Robust tabu search for the quadratic assignment problem, Parallel Comput. 17 (1991) 443–455.
[27] Z. Drezner, The extended concentric tabu search for the quadratic assignment problem, Eur. J. Oper. Res. 160 (2005) 416–422.
[28] A. Misevicius, A tabu search algorithm for the quadratic assignment problem, Comput. Optim. Appl. 30 (1) (2005) 95–111.
[29] T. James, C. Rego, F. Glover, Multistart tabu search and diversification strategies for the quadratic assignment problem, IEEE Trans. Evol. Comput. 39 (2009) 579–596.
[30] Z. Drezner, A new genetic algorithm for the quadratic assignment problem, Inf. J. Comput. 15 (3) (2003) 320–330.
[31] A. Misevicius, Genetic algorithm hybridized with ruin and recreate procedure: application to quadratic assignment problem, Knowl. Based Syst. 16 (5-6) (2003) 261–268.
[32] A. Misevicius, An improved hybrid genetic algorithm: new results for the quadratic assignment problem, Knowl. Based Syst. 17 (2-4) (2004) 65–73.
[33] Z. Drezner, Compounded genetic algorithms for the quadratic assignment problem, Oper. Res. Lett. 33 (2005) 475–480.
[34] T. Stützle, M. Dorigo, ACO algorithms for the quadratic assignment problem, in: D. Corne, M. Dorigo, F. Glover (Eds.), New Ideas in Optimization, McGraw-Hill, 1999, pp. 33–50.
[35] M. Lopez-Ibanez, L. Paquete, T. Stützle, On the design of ACO for the biobjective quadratic assignment problem, ANTS 2004, LNCS 3172 (2004) 214–225.
[36] A. Acan, An external partial permutations memory for ant colony optimization, EvoCOP 2005, in: LNCS 3448, 2005, pp. 1–11.
[37] Y. Li, P.M. Pardalos, M.G.C. Resende, A greedy randomized adaptive search procedure for the quadratic assignment problem, in: P.M. Pardalos, H. Wolkowicz (Eds.), Quadratic Assignment and Related Problems, DIMACS 16, Amer. Math. Soc., 1994, pp. 237–261.
[38] H. Li, D. Landa-Silva, An elitist GRASP metaheuristic for the multi-objective quadratic assignment problem, in: Ehrgott M., et al. (Eds.), Evolutionary Multiobjective Optimization EMO-2009, LNCS 5467, 2009, pp. 481–494.

[39] P. Merz, B. Freisleben, A comparison of memetic algorithms, tabu search, and ant colonies for the quadratic assignment problem, IEEE Trans. Evol. Comput. 4 (2000) 337–352.
[40] D. Garret, An empirical comparison of memetic algorithm strategies on the multiobjective quadratic assignment problem, in: IEEE Symposium on Computational Intelligence in Multicriteria Decision Making, 2009.
[41] Z. Drezner, Tabu search and hybrid genetic algorithms for quadratic assignment problems, in: W. Jaziri (Ed.), Tabu Search, InTech, 2008, pp. 89–108.
[42] A. Misevicius, E. Guogis, Computational study of four genetic algorithm variants for solving the quadratic assignment problem, in: T. Skersys, R. Butleris, R. Butkiene (Eds.), International Conference on Information and Software Technologies – ICIST 2012, 2012, pp. 24–37.
[43] T. Stützle, Iterated local search for the quadratic assignment problem, Eur. J. Oper. Res. 174 (2006) 1519–1539.
[44] R.K. Ahuja, K.C. Jha, J.B. Orlin, D. Sharma, Very large-scale neighborhood search for the quadratic assignment problem, Inf. J. Comput. 19 (2007) 646–657.
[45] H. Liu, A. Abraham, J. Zhang, A particle swarm approach to quadratic assignment problems, in: A. Saad, K. Dahal, M. Sarfraz, M. Roy (Eds.), Soft Computing in Industrial Applications, Springer, 2007, pp. 213–222.
[46] T. El-Cheikh, Adapting Particle Swarm Optimization Algorithm to Solve Quadratic Assignment Problems, Doctoral dissertation, University of Louisville, 2010.
[47] D. Davendra, I. Zelinka, G. Onwubolu, Clustered population differential evolution approach to quadratic assignment problem, in: Proc. of IEEE Conf. on Evolutionary Computation – CEC, 2009, 2009, pp. 1224–1231.
[48] J.I. Kushida, Solving quadratic assignment problems by differential evolution, in: Proceedings of Int. Conf. on Soft Computing and Intelligent Systems (SCIS), 2012, pp. 639–644.
[49] A.S. Mamaghani, M.R. Meybodi, An application of imperialist competitive algorithm to solve the quadratic assignment problem, in: Int. Conf. on Internet Technology and Secured Transactions (ICITST), 2011, pp. 562–565.
[50] E. Duman, M. Uysal, A.F. Alkaya, Migrating birds optimization: a new metaheuristic approach and its performance on quadratic assignment problem, Inf. Sci. 217 (2012) 65–77.
[51] A.Y.S. Lam, V.O.K. Li, Chemical-reaction-inspired metaheuristic for optimization, IEEE Trans. Evol. Comput. 14 (3) (2010) 381–399.
[52] K. Li, Z. Zhang, Y. Xu, Chemical reaction optimization for heterogeneous computing environments, in: IEEE Intl. Symposium on Parallel and Distributed Processing, 2012, pp. 17–23.
[53] T.K. Truong, K. Li, Y. Xu, Chemical reaction optimization with greedy strategy for the 0/1 knapsack problem, Appl. Soft Comput. 13 (2013) 1774–1780.
[54] Y. Xu, K. Li, L. He, T.K. Truong, A DAG scheduling scheme on heterogeneous computing systems using double molecular structure-based chemical reaction optimization, J. Parallel Distrib. Comput. 73 (2013) 1306–1322.
[55] A.M. Geoffrion, G.W. Graves, Scheduling parallel production lines with changeover costs: practical application of quadratic assignment/LP approach, Oper. Res. 24 (4) (1976) 595–610.
[56] Z. Tang, M. Liu, K. Li, Y. Xu, A mapreduce-enabled scientific workflow framework with optimization scheduling algorithm, in: IEEE Intl. Conf. on Parallel and Distributed Computing, Application and Technologies, 2012, pp. 599–604.
[57] K. Li, S. Li, Y. Xu, A DAG task scheduling scheme on heterogeneous computing systems using invasive weed optimization algorithm, in: IEEE Intl. Symposium on Parallel Architectures, Algorithms and Programming, 2014, pp. 262–267.
[58] Y. Xu, K. Li, J. Hu, K. Li, A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues, Inf. Sci. 270 (2014) 255–287.
[59] F. Glover, Tabu Search – Part 1, ORSA J. Comput. 1 (2) (1989) 190–206.
[60] F. Glover, Tabu Search – Part 2, ORSA J. Comput. 2 (1) (1990) 4–32.
[61] F. Glover, A template for scatter search and path relinking, in: LNCS 1363, 1998, pp. 13–54.
[62] A. Misevicius, B. Kilda, Comparison of crossover operators for the quadratic assignment problem, Inf. Technol. Control 34 (2) (2005) 109–119.
[63] F. Glover, Genetic algorithms and scatter search: unsuspected potential, Stat. Comput. 4 (1994) 131–140.
[64] A. Acan, A. Unveren, Evolutionary multiobjective optimization with a segment based external memory support for the multiobjective quadratic assignment problem, in: Congress on Evolutionary Computation – CEC, 2005, pp. 2723–2729.
[65] Anon, http://www.opt.math.tu-graz.ac.at/qaplib/inst.html, OR http://www.seas.upenn.edu/qaplib/inst.html
[66] Anon, http://mistic.heig-vd.ch/taillard/problemes.dir/qap.dir/qap.html
[67] Anon, http://business.fullerton.edu/isds/zdrezner/programs.htm
[68] T. James, C. Rego, F. Glover, A cooperative parallel tabu search algorithm for the quadratic assignment problem, Eur. J. Oper. Res. 195 (2009) 810–826.
[69] L. Tseng, S. Liang, A hybrid metaheuristic for the quadratic assignment problem, Comput. Optim. Appl. 34 (1) (2006) 85–113.
[70] Z. Drezner, P.M. Hahn, E.D. Taillard, A Study of Quadratic Assignment Problem Instances that are Difficult for Metaheuristic Methods, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.8.4479
[71] U. Tosun, On the performance of parallel hybrid algorithms for the solution of the quadratic assignment problem, Eng. Appl. Artif. Intell. 39 (2015) 267–278.
[72] U. Benlic, J.K. Hao, Memetic search for the quadratic assignment problem, Expert Syst. Appl. 42 (2015) 584–595.
[73] T. Dokeroglu, Hybrid teaching-learning based optimization algorithms for the quadratic assignment problem, Comput. Ind. Eng. 85 (2015) 86–101.