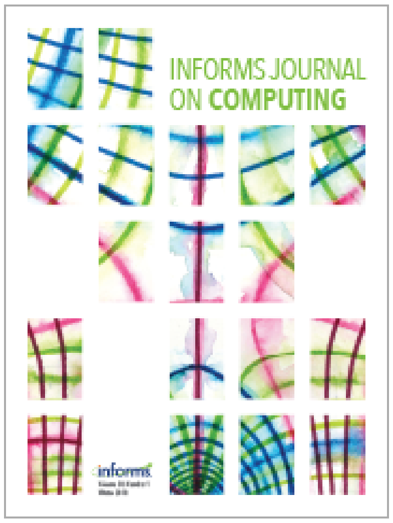

Randomized Decomposition Solver with the Quadratic Assignment Problem as a Case Study

Krešimir Mihić, Kevin Ryan, Alan Wood

http://pubsonline.informs.org/journal/ijoc/

INFORMS JOURNAL ON COMPUTING
Vol. 30, No. 2, Spring 2018, pp. 295–308
ISSN 1091-9856 (print), ISSN 1526-5528 (online)

Randomized Decomposition Solver with the Quadratic Assignment Problem as a Case Study

Krešimir Mihić,[a] Kevin Ryan,[b] Alan Wood[a]

[a] Oracle Labs, Belmont, California 94002; [b] H. Milton Stewart School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, Georgia 30332

Contact: kresimir.mihic@oracle.com, http://orcid.org/0000-0001-7896-2427 (KM); kevin.ryan30@gmail.com (KR); alan.wood@oracle.com (AW)

Received: January 4, 2016
Revised: January 31, 2017; June 7, 2017; August 28, 2017
Accepted: August 31, 2017
Published Online: April 23, 2018

https://doi.org/10.1287/ijoc.2017.0781

Abstract. This paper presents a new local search approach, called randomized decomposition (RD), for solving nonlinear, nonconvex mathematical programs. Starting from a feasible solution, RD partitions the problem's decision variables into a randomly ordered list of randomly generated subsets. RD then optimizes over the variables in each subset, keeping all other variables fixed. Unlike most other decomposition methods, no knowledge of the problem structure is required. RD has been combined with a metaheuristic RDPerturb, for escaping local optima, to create a generic framework for solving mathematical programs, especially hard combinatorial nonconvex problems. The framework has been implemented as an optimization platform we call RDSolver and successfully applied to over 400 instances of the quadratic assignment problem (QAP). The results obtained by RDSolver are competitive with the solutions obtained by heuristics specially tailored for those problems, even though RDSolver is a general purpose mathematical programming solver. In addition to a strong performance on previously solved problems, RDSolver has found two new best known solutions and provided solutions to 68 large QAP problems for which no solutions have been previously reported.

History: Accepted by Alice Smith, Area Editor for Heuristic Search.
Supplemental Material: The online supplement is available at https://doi.org/10.1287/ijoc.2017.0781.

Keywords: integer programming • heuristic algorithms • nonlinear programming

1. Introduction

Customers in industrial settings are interested in mathematical optimization for business applications such as portfolio optimization, vehicle routing, price discount optimization, and shelf-space optimization. In these applications, the decision variables are often items (nonnegative integers) with a limited number of choices or prices with a limited selection of values called a price ladder, e.g., $\{\$1.49, \$1.79, \$1.99, \$2.29, \$2.49\}$. There can be many constraints that express item limits and interactions, finite amount of store or warehouse storage and display space, discount limitations, time horizons, and many other factors. The objective function, which is usually to maximize revenue or profit or to minimize cost, can be highly nonlinear because of complex interactions such as price elasticity. For example, a grocery store might choose to discount a particular type of cereal, which should increase the amount of that cereal purchased and may decrease the amount of other types of cereal purchased, but could also lead to increased sales of milk and sugar. The relationship of these econometric substitutes and complements are often expressed as complex exponential functions.

These kinds of business problems lead to combinatorial mathematical optimization problems, which can in principle be solved by complete enumeration of all possible decision variable value combinations. In practice, of course, there are far too many combinations for complete enumeration to be viable, so techniques such as branch and bound, tabu search, and many kinds of genetic algorithms have been applied to these problems. However, in our testing across several types of these real-world business problems, no single approach seemed to solve all of them to our satisfaction, and we wanted a single approach to minimize the amount of software we needed to develop and maintain. The idea of using enumeration led us to consider a different solution approach—decomposing the problem into subproblems that are small enough to be solved optimally by enumeration or another strategy. Because we had a wide variety of problems to solve, problem structure could not be used to guide the decomposition process as it is done in classic approaches (say Bender's or Dantzig-Wolfe decomposition). In contrast to the classic approaches, where constraints/variables are left out of the optimization and "priced in" if they are violated (constraints) or beneficial (variables), we are performing a variant of a local search. We randomly decompose the decision variables into subsets and optimize each of these subproblems in random order with the decision variables outside a subproblem held constant while optimizing that subproblem. We call this search technique randomized decomposition (RD). When combined with a solution perturbation method called RDPerturb to escape local optima, RD seems to work well for the combinatorial optimization problems inherent in our products.

This paper presents RD and the complete solution framework, called RDSolver, that combines RD with RDPerturb to create a general purpose, nonlinear, nonconvex discrete solver for solving hard, real-world mathematical programs. RD is a novel decomposition algorithm that essentially partitions the solution space into random subspaces and finds a local optimum in each subspace independently.

RDSolver has been successfully applied to a wide range of problems, including revenue management problems, graph problems (min/max cut, partitioning, and clustering), the traveling salesman problem, and assignment problems. It is currently shipping in Oracle products for project portfolio optimization and vehicle routing. It is also being integrated into additional Oracle products for discount optimization and assortment optimization and is under evaluation for use in other areas (Oracle Labs 2016).

In this paper we give an assessment of RDSolver performance when addressing a classical optimization problem, the quadratic assignment problem (QAP) by comparing it with the best performing heuristic solutions. The QAP is an assignment-type optimization problem, commonly formulated as the assignment of a set of n facilities to a set of n locations. For each pair of locations, a distance is specified and for each pair of facilities a weight or flow is specified (e.g., the amount of supplies transported between the two facilities). The problem is to assign all facilities to different locations with the goal of minimizing the sum of the distances multiplied by the corresponding flows.

RDSolver has been implemented and successfully applied to over 400 instances of the QAP. Typically, for the QAP problems tested, RDSolver quickly finds local optima that are within a fraction of a percent of the best known solution and converges to the best known solution in an execution time comparable to that of other QAP solution techniques. Numerical experiments done on a wide variety of benchmark instances indicate the ability of RD to solve many different types of QAP problems. Benchmark instances include QAPLIB, described in Burkard et al. (1997), and hard problems of large size, described in Drezner et al. (2005). The results obtained by RDSolver are competitive with the solutions obtained by heuristics specially tailored for QAP problems, even though RDSolver is a general purpose mathematical programming solver. In addition to a strong performance on previously solved problems, RDSolver has found two new best known solutions and provided solutions to 68 large QAP problems for which no solutions have been previously reported (Oracle Labs 2016).

The paper is organized as follows. RDSolver is described in Section 2, including both the general methodology of randomized decomposition (RD) in Section 2.1 and the perturbation method, RDPerturb, used to escape a local optimum in Section 2.2. Section 2.3 explains the motivation for RD. Section 3 describes the QAP and the application of RD to QAP problems. In Section 4, the performance of RD is compared with the best existing methods for solving QAP problems. In Section 5, we compare QAP problem instances for which RDSolver performed well and poorly, and develop a heuristic that could be used to improve performance for various types of problems. Conclusions are presented in Section 6. The online supplement contains examples of applying RD to other classic optimization problems and a proof of the heuristic property presented in Section 5.

2. RDSolver

RDSolver combines a large neighborhood search (LNS) metaheuristic for exploring the solution space with a randomized decomposition (RD) approach to find a local optimum. The LNS metaheuristic, RDPerturb, is similar to many LNS algorithms in the sense that it attempts to escape a local optimum by changing a large number of the current decision variable values. The decision variables to be changed are selected at random, but the selected random decision variable values are required to remain in the set of feasible solutions, unlike other LNS algorithms that may select a completely random and possibly infeasible starting point. The combination of using a decomposition approach to search for a local optimum, together with a perturbation method to escape the current neighborhood, can be characterized as iterative decomposition.

The pseudo code of RDSolver is presented in Algorithm 1. The algorithm starts with an initial feasible solution (line 1), constructed by any applicable means, followed by RD (Algorithm 2) to find a local optimum in the neighborhood of the initial solution (line 2). This local optimum is then subjected to improvements using RDPerturb (Algorithm 3) and RD until the termination criteria are satisfied (lines 6–21). In essence, the algorithm is RD(RDPerturb(x)), that is, use RDPerturb to try to escape from the current local optimum (line 12) and then use RD to find the local optimum around the perturbed point (line 7). The number of unsuccessful trials to improve the solution, *nTrials*, is used by RDPerturb to decide on the magnitude of the next permutation as described in Section 2.2. The *rsFlag* indicates whether RDPerturb performed a small perturbation or a large perturbation (*Restart*). After a restart, and until a new optimum is found, RDPerturb is run in the neighborhood of the large perturbation (line 9) instead of the current best solution (line 15) as it is when a small perturbation has been performed.

Algorithm 1 (RDSolver)

```
Require: Problem description (variables $x$, cost
            function $cost(x)$, constraints)
         Maximum allowed runtime, $\max_{RT}$
Ensure: Best solution found, $x_{best}$
 1: Construct an initial feasible solution $\hat{x}$.
 2: $x_{best} \leftarrow RD(\hat{x})$
 3: $x_{start} \leftarrow x_{best}$, $x_{candidate} \leftarrow x_{best}$
 4: $nTrials \leftarrow 0$            // $nTrials$ is used in RDPerturb
 5: $rT \leftarrow 0$                      // $rT$ is current runtime
 6: while $rT < \max_{RT}$ and other termination
      criteria not satisfied[1]
 7:   $\{x_{candidate}, rsFlag\} \leftarrow RDPerturb(x_{start}, nTrials)$
                                   // Described in Section 2.2
 8:   if $rsFlag = true$ then
 9:     $x_{start} \leftarrow x_{candidate}$
10:     $nTrials \leftarrow 0$
11:   end if
12:   $x^* \leftarrow RD(x_{candidate})$
                                   // Described in Section 2.1
13:   if $cost(x^*) < cost(x_{best})$ then
14:     $x_{best} \leftarrow x^*$
15:     $x_{start} \leftarrow x_{best}$
16:     $nTrials \leftarrow 0$
17:   else
18:     $nTrials \leftarrow nTrials + 1$
19:   end if
20:   $rT \leftarrow rT +$ time spent executing steps 7–19
21: end while
22: return $x_{best}$.
```

2.1. RD

The purpose of local neighborhood search is to find a local optimum. Many algorithms use steepest descent or gradient-based methods to perform this search, e.g., Benlic and Hao (2015). Our approach to local neighborhood search is a decomposition method called RD. A decomposition method is a multistage approach in which a problem is partitioned into smaller problems that can be solved independently.

RD creates the subproblems by partitioning the problem's decision variables into subsets following some probability distribution or an algorithm based on the problem's domain knowledge. Each of the subproblems is optimized with the variables outside the subset held constant. This leads to simpler problems to solve by lowering the dimensionality of the problem. Adding domain knowledge, if available, can lead to even simpler problems or easier-to-solve subproblem formulations.

Algorithm 2 (RD)

```
Require: Feasible solution $\hat{x}$, problem definition
            (cost function $cost(x)$, constraints)
         Maximum number of trials with no
           improvement, $nTrials_{RD}$
         Min subproblem size, $k$
         Problem size (number of decision variables), $n$
Ensure: Local optimum solution, $x$
 1: $x \leftarrow \hat{x}$
 2: $cnt \leftarrow 0$
 3: X := {set of all decision variables
           $x_i$: $i \in \mathcal{Z}, 1 < i \le n$}
 4: C := {set of all constraint functions}
 5: while $cnt < nTrials_{RD}$
    reshuffle:
 6:   $j \leftarrow 1$
 7:   $bf \leftarrow false$
 8:   while $\bigcup_{i=1}^{j} P_i \ne X$
 9:     $S_j \leftarrow \varnothing$, $P_j \leftarrow \varnothing$
10:     while $|S_j| < k$ or $\bigcup_{i=1}^{j} S_i \ne X$
11:       Select $x_y$ randomly from $X \setminus \bigcup_{i=1}^{j} P_i$
12:       $P_j \leftarrow P_j \cup \{x_y\}$
13:       $S_j \leftarrow S_j \cup \{x_y\}$
14:       for each active constraint $c_i \in C$ where
            variable $x_y$ is an operand do
15:         T := {set of variables that are
                    operands of $c_i$}
16:         Select $x_z$ from $T \setminus S_j$
17:         $S_j \leftarrow S_j \cup \{x_z\}$
18:       end for each
19:     end while
20:     Create a subproblem from the original
          problem by considering decision variables
          $x_i \notin S_j$ as constants
21:     Solve the subproblem, i.e., find the values
          $x_i^* \in S_j$, that minimize $cost(x^*)$ [2]
22:     if $cost(x^*) cost(x)$ then
23:       $x \leftarrow x^*$.
24:       $bf \leftarrow true$
25:     end if
26:     $j \leftarrow j + 1$
27:   end while
28:   if $bf = true$ then
29:     $cnt \leftarrow 0$
30:   else
31:     $cnt \leftarrow cnt + 1$
32:   end if
33: end while
34: return $x$.
```

RD pseudocode is shown in Algorithm 2.[3] The outer **while** loop in Algorithm 2 (lines 5–33) describes the process of creating and solving subproblems, which we call a *reshuffle*. The number of reshuffles is controlled by an input parameter $nTrials_{RD}$ (line 5), which defines how many times RD is allowed to seek for a better solution without any improvement in the value of the objective (cost) function. Creating the subset of variables that defines the subproblem is shown in lines 10–27. The simplest approach for creating subsets is to simply choose variables at random.[4] This works in many situations, but a problem with choosing variables at random is that the variable may be part of an active constraint,[5] and thus have its value fixed. If there are many active constraints, the variables may not have enough freedom to improve the solution. Therefore, we use a method for creating subsets, shown in lines 14–18, that adds enough additional variables to the subset to allow the selected variable to change its value. To allow the selected variable x_y to change its value, we choose a random variable from all its active constraints and add it to the subset S_j. This allows us to change the value of x_y by also changing the values of variables in all its active constraints. An example of using this method to create subproblems is given in the online supplement.

The approach in lines 10–19 can be modified if we are using domain knowledge to better group variables into subsets. At least one variable is chosen at random, and the rest of the variables are then added in a semi-random fashion based on domain knowledge. The variable selection can be specialized to give preference to variables of larger (smaller) values, or variables that are grouped around some value, or variables that have wide variations, and so forth. The min-cut problem described in the online supplement is an example of using domain knowledge to group variables into subproblems as is the quadratic assignment problem described in Section 3.

Although not required by the algorithm, in practice we create subproblems with approximately the same number of decision variables. These subproblems can be solved by any applicable algorithm (line 21), including enumeration for small subproblems (our default approach), simplex or interior-point methods using third party solvers if applicable, or heuristics such as Lin-Kernighan heuristic (LKH) for the traveling salesman problem (Helsgaun 2000). They can also be solved by recursive application of RD to make even smaller subproblems to solve via enumeration or other methods. In the limit, recursive application of RD creates subproblems with a single decision variable for which the solution is trivial.

2.2. RDPerturb

The large neighborhood search function, RDPerturb, performs the decision variable modification shown in Algorithm 3. It chooses the decision variables to modify and updates them with randomly chosen values, subject to feasibility constraints. The decision variable modification algorithm usually employed is probabilistic search, which searches in a neighborhood of the current best solution. If there has been no improvement for several iterations of probabilistic search ($nTrials = nTrials_{\max}$), a large perturbation to the current solution, called a restart, is performed. While many different perturbation strategies exist, probabilistic search and restart seemed to be the most efficient for RDSolver in our experimentation.

Algorithm 3 (RDPerturb)

```
Require: Maximum number of iterations with no
            improvement before restart, $nTrials_{\max}$
         Number of variables to change for the
           restart method, $r_n$, $r_n \le n$
         Current number of trials, $nTrials$
         Starting (feasible) solution $x_{\text{start}}$, constraints
         Probabilistic search distribution parameters:
           $ps_{\min}, ps_{\max}, \lambda$
Ensure: A new candidate solution, $x_{\text{candidate}}$
          and restart flag, $rsFlag$
 1: X := {set of decision variables $x_i$: $i \in \mathcal{I}, 1 < i \le n$}
 2: if $nTrials < nTrials_{\max}$ then
      Probabilistic search:
 3:   Draw random number $y$ from f$(y, ps_{\min}, ps_{\max})$
                                          // Equation (1)
 4:   $m \leftarrow \lfloor y \rceil$
 5:   V := {set of decision variables $x_i \in$ X, chosen
            by drawing from some probability
            distribution or by an algorithm based on
            problem's domain knowledge}, $|V| \le m$
 6:   F := {set of all feasible solutions
            in proximity of $x_{\text{start}}$ found by considering
            decision variables $x_i \notin$ V as constants}[6]
 7:   Select $x_{\text{candidate}}$ randomly from F
 8:   $rsFlag \leftarrow false$
 9: else
      Restart:
10:   S ← ∅
11:   $x_{\text{candidate}} \leftarrow x_{\text{start}}$
12:   while |S| < $r_n$
13:     Select $x_i \in$ X\S at random
14:     S ← S ∪{$x_i$}
15:     F := {set of all values $v_j$ of decision variable $x_i$
             for which the problem remains feasible}[6]
16:     Select $v_j \in$ F at random
17:     $(x_{\text{candidate}})_i \leftarrow v_j$
18:     $rsFlag \leftarrow true$
19:   end while
20: end if
21: return {$x_{\text{candidate}}, rsFlag$}.
```

***Probabilistic search* (*PS*).** The purpose of probabilistic search, shown in Algorithm 3, lines 3–8, is to explore the search space with a bias toward searching near the current best solution. To perform this search, we select a range of decision variables to modify, $(ps_{\min}, ps_{\max})$ with $1 \le ps_{\min} < ps_{\max} \le n$, where n is the total number of decision variables (problem size), and then select a random number of decision variables m to modify within that range. Any distribution function could be used to select m, but empirically we found that a distribution that favored values near $ps_{\min}$ while occasionally selecting a value near $ps_{\max}$ was the most efficient. The approach that seemed to work best was to select a random value from the truncated exponential density function in Equation (1) and then round that value to the nearest integer.

$$f(y, ps_{\min}, ps_{\max}) = \frac{\lambda e^{-\lambda y}}{e^{-\lambda ps_{\min}} - e^{-\lambda ps_{\max}}}. \tag{1}$$

After selecting m on lines 3 and 4, the set S of m decision variables to be modified is created (line 5) by selecting m random variables x_i from the set of all variables X. The new values of the selected variables are chosen (line 7) from set F, which is the set of all possible variable values that define feasible solutions around solution x_{start} (line 6).

Probabilistic search exploration can be controlled by modifying the parameters $ps_{\min}$, $ps_{\max}$, and λ. For the QAP, the best performance was obtained by setting $ps_{\max} = n$, $ps_{\min}$ to 10% of $ps_{\max}$ and λ to $\ln(2)/ps_{\min}$, which means that a random value of $2ps_{\min}$ is half as likely as $ps_{\min}$. We have found that different parameter values work better for other combinatorial optimization problems. We even found that a different λ worked better for one class of QAP problems (see Table 1).

Other search space exploration strategies we tried did not seem to work as well as probabilistic search. For example, we tried a bread-first search strategy of randomly selecting and modifying a few decision variables, then repeating if the solution did not improve. If the solution had not improved after several trials, we would increase (e.g., double) the number of decision variables to modify. We also tried a depth-first search strategy, but probabilistic search found better solutions faster than all the other strategies we tried.

Restart. The purpose of restart is to produce a candidate solution that is far away from the current best solution, meaning that we randomly change a large number of decision variable values. Our restart approach is shown in lines 10–19 of Algorithm 3. All, or a large subset ($r_n \le n$), of decision variables are chosen in random order (lines 13–17), and for each variable, a new value is randomly selected from the set of values for which the solution remains feasible (line 16). The drawback of this approach is that many variable values may remain unchanged because they only have a single value for which the solution remains feasible, e.g., if there are active constraints. In that case, we can use the same approach that is taken in lines 14–18 of Algorithm 2 to provide freedom for the variable values to change.

2.3. Randomized Decomposition Rationale

Many optimization algorithms such as simulated annealing and tabu search incorporate an element of randomness in an attempt to avoid being stuck in a local optimum. However, in our approach, we seem to have taken randomness to an extreme. An optimization is performed around every random feasible point in the search space selected by the RDSolver algorithm, even if it may return the same local optimum. No history of search space exploration is kept, so there is no way to avoid portions of the search space that appear fruitless or that have been adequately searched in the past. Why would we propose such an unintelligent algorithm?

RD was initially targeted at real-world revenue management problems, where the speed of getting an improved solution (compared to some initial solution) is more valuable than a slightly better solution obtained after a long execution time. The decision variables are usually items (nonnegative integers) with a limited number of choices and limited selection of prices. There are lots of constraints that express item limits and interactions, finite amount of store or warehouse storage and display space, and other limitations. The objective function is often nonlinear and very complex due to the econometric of substitutes and complements. Standard third-party solvers had difficulty with the nonlinearity of the objective function and the noncontinuity of the decision variables. Because the problem had no structure to use for decomposition and because randomization is a powerful tool in many contexts, we tried random decomposition with good success. Having had good success with RD on a limited class of problems, we wondered if it could

Table 1. Control Parameters and Default Values for RDSolver

Parameter	Description	Value
n	Problem size (number of decision variables)	Variable
k	No. of variables for subproblems (RD)	3
$nTrials_{\text{RD}}$	Max. no. of trials with no improvement (RD)	n
$nTrials_{\max}$	Max. no. of trials with no improvement before *restart* (RDPerturb)	$5n$
$ps_{\min}, ps_{\max}$	Probabilistic search parameters (RDPerturb)	$0.01n, n$
$\ln(2)/\lambda$	The mean of the exponential distribution (RDPerturb)	$0.1n, 0.3n$[*]
$\max_{\text{RT}}$	Max. allowed execution time (RDSolver)	2 h[†]

[*]$0.3n$ is used for QAPLIB type III instances only.
[†] For all but large tai*xx*e*yy* instances.

be generalized and applied to classic combinatorial problems such as the QAP, min/max cutsets, facility location, the traveling salesman problem (TSP), and so forth. Our preliminary testing has shown excellent results on these classic problems, with the QAP results highlighted in this paper and results for other classic problems presented in the online supplement.

The combinatorial optimization problems of interest to our business are NP-hard, but finite, so they could in principle be solved by complete enumeration. Heuristics for these kinds of problems may be inspired by biology (genetic algorithms and tabu search), nature (ant colony algorithms), and physical processes (simulated annealing). RD is inspired by Moore's Law. The relentless increase in microprocessor performance allows enumeration to be feasible for problems with billions of combinations. Although our problems are often much larger, we seek to combine evaluation of a massive number of potential solutions with standard large neighborhood search strategies for escaping local optima instead of spending compute cycles on more clever local neighborhood search strategies. It is hard to make theoretical justifications for superior performance of a certain heuristic, but our experimental results indicate the promise of this approach.

The seeming weaknesses of RD turn out to be strengths for many problems. For large problems, RD can evaluate thousands of points in the local neighborhood in the same time as a single calculation of a conjugate gradient or Hessian-based search direction. Not worrying about search history means no data to share and no need for synchronization. Thus, the evaluation of RDPerturb-selected points can all be done in parallel, with maximal use of hardware resources, which perfectly matches the current massive multithreading trend in computer architecture. Another positive feature of RD is that it has a small memory footprint. Because RD doesn't keep search history, all RDSolver needs to keep in memory is the objective function and constraint coefficients, along with the current values of the decision variables and objective function. For example, it does not need to keep items such as parents and offspring like some genetic algorithms or a "tabu list" like tabu search.

As previously mentioned, the subproblems generated by RD can be solved by any method. Our default approach is to randomly select a small subset of decision variables and optimize the decision variable values in this subset by enumeration, essentially a brute-force exploration of the solution space. For each candidate solution (possible decision variable values), the objective function is evaluated, followed by evaluation of constraint functions. The order can be reversed if the objective function is expensive to evaluate compared to the effort needed to evaluate the constraints. Note that for assignment-type combinatorial optimization problems, such as the QAP, the constraints are a 1-1 assignment of facilities to locations. These constraints are automatically satisfied by solution construction and do not have to be checked.

Our current RDSolver implementation combines RD with RDPerturb and solves the subproblems created by RD using enumeration (or recursive application of RD followed by enumeration). However, the RD and RDPerturb algorithms could be useful in other settings. RDPerturb could be used with a different local neighborhood search technique. RD could be combined with other perturbation or large neighborhood search methods. The subproblems created by RD could be solved with a local neighborhood search technique other than enumeration. We have experimented with other combinations, such as using LKH-5 (Helsgaun 2000) in place of RD for the traveling salesman problem, but thus far the improved performance does not justify the cost of maintaining a more complex product code base.

3. The Quadratic Assignment Problem

3.1. Formulation and History

The quadratic assignment problem (QAP) belongs to a class of combinatorial optimization problems that arise from problems of practical interest. The QAP objective is to assign n facilities to n locations in such a way as to minimize the assignment cost. The assignment cost is the sum, over all pairs, of a weight or flow between a pair of facilities multiplied by the distance between their assigned locations. The QAP was first proposed by Koopmans and Beckmann (1957) in a context of the location of indivisible economic quantities such as capital equipment. Since then, researchers have developed numerous exact and heuristic solutions that address real-world problems such as electrical circuit wiring/routing, campus planning, hospital layout, warehouse management and distribution strategies, ordering of correlated data in magnetic medium, image processing, and others as described in Cela (1998).

Mathematically, the QAP can be presented as follows: given a set $\Pi(n)$ of all permutations of n elements and two $n \times n$ matrices, **A** and **B**, find a permutation π^* that minimizes the quantity

$$C(\pi) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{i,j} b_{\pi(i),\pi(j)}, \qquad \pi \in \Pi(n). \tag{2}$$

As shown in Sahni and Gonzalez (1976), the QAP is NP-hard and no ϵ-approximation algorithm exists unless $P = NP$. In contrast to some other NP-hard combinatorial problems (say, the traveling salesman problem), QAP instances of larger sizes ($n > 40$) are considered to be intractable and cannot be solved exactly (though some instances of large size have been solved, e.g., esc128). Currently, the only practical solutions for solving large QAP instances are heuristic methods.

The first heuristic algorithm was a descent algorithm named CRAFT, developed by Armour and Buffa (1963). More recent algorithms use metaheuristics such as tabu search (e.g., Drezner 2005, James et al. 2009), simulated annealing (e.g., Kovac 2013), ant colony systems (e.g., Ramkumar et al. 2009), genetic algorithms (e.g., Drezner 2008b), specially designed heuristics (e.g., Stutzle 2006) and a combination of heuristics for local and large neighborhood search (e.g., Benlic and Hao 2013, 2015). For a more complete discussion of QAP solutions, see Cela (1998), Loiola et al. (2007), and Burkard (2013).

The current algorithms with the best results on QAPlib and other problems are breakout local search (BLS) in Benlic and Hao (2013), BMA (population-based memetic algorithm) in Benlic and Hao (2015), and CPTS (cooperative parallel tabu search algorithm) in James et al. (2009). The RDSolver results are compared with the results from these algorithms in Section 4. BLS uses a descent procedure to discover local optima and an adaptive perturbation mechanism that does small jumps in the solution space and then a large jump if the solution is not improving. The small jumps alternate between jumps based on tabu search that minimize cost degradation when swapping assignments, jumps that favor least recently performed swaps, and random swaps. The small jumps are similar in spirit to RDPerturb (described in the Section 2.2), with the difference that RDPerturb uses only the probabilistic approach for choosing the variables and does not only swap the values between two variables (note that swapping can be done if we require subproblem size to be exactly 2). The big jumps in both algorithms are similar as well, with the only difference, again, being the number of variables chosen at a time.

The population-based memetic algorithm (BMA) is a genetic algorithm that combines BLS for local search with uniform crossover for creating children, a fitness-based pool updating approach, and an adaptive mutation procedure. Cooperative parallel tabu search (CPTS) executes multiple tabu search operators in parallel. The stopping criterion and tabu tenure parameters are varied among the parallel executions, and information is exchanged among the searches for intensification and diversification of solutions. BMA and CPTS belong to a completely different class of heuristic approaches with no similarity to RD and RDPerturb.

Many heuristic QAP methods have a major limitation: the algorithm is tailored to a specific QAP problem structure. One heuristic (or its variant) is excellent for solving one type of problem but does not work well for others. For example, in Drezner (2008a) the author describes five different tabu search techniques used as improvement procedures for nine different hybrid genetic algorithms (and a number of variants thereof). Individually, these algorithms produce the best known results for a few QAP problems with a specific structure, but a single heuristic algorithm alone cannot be used over a wide range of QAP problems. The latter limitation seriously limits the applicability of the algorithms for real-world problems, where solution surfaces are usually unknown and/or change over time. In addition, most heuristic QAP methods cannot be applied to other optimization problems such as TSP, and we are interested in developing a method that works well across a broad range of combinatorial optimization problems.

3.2. Randomized Decomposition Solution for the QAP

3.2.1. Setup. The QAP is solved by RDSolver, where the subproblems in the local neighborhood search are permutations of the selected variable values. This permutation representation enables us to encode the uniqueness constraint into the problem itself, eliminating the need for special handling of feasibility regions (Algorithm 2, lines 16–19). The reshuffling operation is performed following the procedure outlined in Algorithm 2, with the following changes:

- *lines* 10–19: $S_j := \{$set of randomly chosen decision variables $x_i \in X\}$, $|S_j| = k$. The constant k specifies the size of a subproblem.
- *line* 21: Solving subproblems is done by evaluating all possible permutations of values of variables $x_i \in S_j$. For example, if we choose $S_j = \{x_5, x_7, x_{11}\}$ and (initially) $x_5 = 10$, $x_7 = 5$ and $x_{11} = 22$, then we evaluate the objective function value for each of the following value assignments: $(x_5, x_7, x_{11}) \in \{(10,5,22), (10,22,5), (5,10,22), (5,22,10), (22,10,5), (22,5,10)\}$.

RDPerturb follows Algorithm 3, with the set S (line 5) defined as $S := \{$set of randomly chosen decision variables $x_i \in X\}$, $|S_j| = k$, and the set F (line 6) defined as $F := \{$set of all possible permutations of values of variables $x_i \in S\}$. The initial feasible solution is selected as a random permutation of variable values.

3.2.2. Complexity. The complexity of the RDSolver algorithm for the QAP problem is primarily the complexity of solving subproblems using RD. For every regrouping of the n decision variables to create a new set of subproblems of size k (called a reshuffle in Algorithm 2), there are $(n/k) \cdot (k! - 1)$ permutations. The additional value of a single permutation ϕ_p, written as $\Delta(\phi_p, V_p)$, is given by

$$\begin{aligned}\Delta(\phi_p, V_p) &= \sum_{i=1}^{n}\sum_{j=1}^{n}(a_{i,j}b_{\phi(i),\phi(j)} - a_{i,j}b_{\pi(i),\pi(j)})\\ &= \sum_{i\in V_p}\sum_{j=1}^{n} a_{i,j}(b_{\phi(i),\phi(j)} - b_{\pi(i),\pi(j)})\\ &\quad - \sum_{i\notin V_p}\sum_{j\in V_p} a_{i,j}(b_{\phi(i),\phi(j)} - b_{\pi(i),\pi(j)}),\end{aligned}$$

where π is the best permutation of the current reshuffling instance, and V_p is the set of permutation locations (i.e., variables) being permuted. Because $|V_p| \le k$, the complexity of a single evaluation of the above equation is at most $O(nk)$ and is repeated $\approx n(k-1)!$ times. Therefore, the total complexity of a single reshuffle is $O(n^2k!)$.

The RDPerturb contribution to the overall complexity is insignificant because all it does is randomly select a relatively small number of decision variables and randomly change the values of the chosen variables to values that produce another feasible solution.

4. Numerical Results

In this section RDSolver is compared with the best known solutions for a wide variety of benchmark QAP problems. The section is divided into two subsections. In Section 4.1, we compare the performance of RDSolver and selected heuristics on QAPLIB instances. In Section 4.2, we address a special kind of very large instances, specially designed to be hard for heuristic solutions. Throughout this section, we compare RDSolver with the best heuristics currently known—BMA, BLS, and CPTS, which were introduced in Section 3.1.

All the experiments were conducted on a Linux-based system (kernel version 3.12.9-201), running Intel Core i5-2520M CPU at 2.5 GHz with 8 GB RAM, using default Java 1.7 settings. Identical RD runtime settings were used across all experiments and instances unless noted otherwise. The default parameter values used are shown in Table 1. All experiments ran using a single thread.

Experimental results summarized in this section were collected by running BLS,[7] BMA,[8] and RDSolver on the same hardware platform to ensure a fair comparison. The CPTS algorithm results were taken from the publication because the source code was not available.

The RDSolver control parameters and corresponding default values are summarized in Table 1. The values were determined empirically. Regardless of the problem type, RDSolver reported similar performance characteristics and seems to be mostly unaffected by a structure, or a lack thereof, of an instance. For a better performance on a specific instance or instance type, one can "tune" RDSolver by adjusting control parameters or by changing the probability distribution function used by RD to generate subproblems. For example, changing the exponential parameter value ($\ln(2)/\lambda$ in Table 1) from $0.1n$ to $0.3n$ causes more variables to be modified during probabilistic search. This improved performance for QAPLIB type III instances that have large valleys (local minima) in the solution space.

4.1. QAPLIB

The QAPLIB benchmark library[9] consists of 135 different problem instances compiled by Burkard et al. (1997). The size of the instances, n, ranges from 12 to 150, and most of the instances with $n \ge 40$ have best known solutions instead of optima, except when the optimum is known by construction. QAPLIB problems are typically classified into four types:

1. real-life instances obtained from practical applications of the QAP;
2. unstructured, randomly generated instances for which the distance and flow matrices are randomly generated based on a uniform distribution;
3. randomly generated instances with structure that is similar to that of real-life instances;
4. instances in which distances are randomly generated on the Manhattan distance grid.

Of the 135 QAPLIB problem instances, 114 are considered easy, meaning that most heuristic algorithms in the last several years have obtained the best known solutions to all of them with a short runtime. RD also obtains the best known solution to these instances, usually within seconds. The performance of RD and the best heuristics for the remaining 21 hard QAPLIB problems is shown in Table 2.

All algorithms were executed 10 times each for each problem instance. The results were compared to the best known solution (BKS), and the average percentage difference $\bar{\delta}$ from the BKS for the 10 runs was calculated. The first three columns show the name of the problem, the number of decision variables, and the BKS. The next two columns show the RD results in terms of $\bar{\delta}$, with the number of times out of 10 the BKS was found in parentheses, and the time in minutes that the algorithm took to complete. The remaining columns show the results for the other instances. The BLS and BMA algorithms were run using the default parameters as described by Benlic and Hao (2013, 2015). The results we obtained by running BLS and BMA differ from the published results, which may be due to running on different hardware. The CPTS results are from James et al. (2009), where they were obtained using 10 (1.3 GHz) Intel Intanium processors running in parallel. Thus, the other methods are at an experimental disadvantage compared to CPTS, but seem to do as well or better.

Although runtimes are provided in many of the tables in this section, comparing RD runtime to the runtime of QAP heuristics should primarily be used for informative purposes. RDSolver is a general-purpose solver for nonlinear, nonconvex problems, and as such implements various structures to handle different data formats, feasibility checks, result correctness verification, cost function calculations, and so forth. This overhead gives RDSolver flexibility to address a wide variety of real-world problems but also puts the solver

Table 2. Comparative Results Between RDSolver and the Three Best Performing QAP Heuristics on Unstructured Instances (Type II), Real-Life Like Instances (Type III), and Grid-Based (Type IV) Instances

			RD		BMA		BLS		CPTS	
Name	n	BKS	$\bar{\delta}$	t [min]	$\bar{\delta}$	t [min]	$\bar{\delta}$	t [min]	$\bar{\delta}$	t [min]
Type II										
tai40a	40	3,139,370	0.25 (1)	49.32	0.09 (0)	20.05	0.04 (4)	25.44	0.15 (1)	3.50
tai50a	50	4,938,796	0.78 (0)	37.83	0.35 (0)	32.39	0.22 (1)	62.91	0.44 (0)	10.30
tai60a	60	7,205,962	0.98 (0)	52.57	0.35 (0)	46.31	0.37 (1)	58.05	0.48 (0)	26.40
tai80a	80	13,499,184	1.20 (0)	41.89	0.56 (0)	86.07	0.55 (0)	71.83	0.69 (0)	94.80
tai100a	100	21,052,466	1.27 (0)	54.52	0.57 (0)	120.35	0.51 (0)	75.64	0.59 (0)	261.20
Type III										
tai50b	50	458,821,517	0.00 (10)	0.22	0.00 (10)	0.49	0.00 (10)	0.43	0.00 (10)	13.80
tai60b	60	608,215,054	0.00 (10)	0.51	0.00 (10)	2.04	0.00 (10)	2.77	0.00 (10)	30.40
tai80b	80	818,415,043	0.00 (10)	22.32	0.00 (10)	12.62	0.00 (10)	11.52	0.00 (10)	110.90
tai100b	100	1,185,996,137	0.00 (10)	27.7	0.03 (7)	44.11	0.00 (10)	26.42	0.00 (9)	241.00
tai150b	150	498,896,643	0.21 (0)	43.2	0.22 (0)	121.12	0.17 (0)	57.22	0.08 (1)	7,377.80
Type IV										
sko72	72	48,498	0.00 (8)	79.94	0.00 (10)	3.35	0.00 (10)	6.29	0.00 (10)	69.60
sko81	81	66,256	0.01 (0)	44.65	0.00 (10)	14.81	0.00 (8)	19.39	0.00 (10)	121.40
sko90	90	90,998	0.01 (2)	34.63	0.00 (9)	25.88	0.00 (10)	41.36	0.00 (10)	193.70
sko100a	100	115,534	0.02 (2)	47.64	0.01 (8)	60.03	0.01 (2)	40.70	0.00 (10)	304.80
sko100b	100	152,002	0.01 (2)	52.18	0.00 (10)	38.67	0.00 (7)	35.25	0.00 (10)	309.60
sko100c	100	147,862	0.00 (2)	43.99	0.00 (10)	23.75	0.00 (9)	39.46	0.00 (10)	316.10
sko100d	100	149,576	0.02 (1)	47.59	0.00 (10)	30.52	0.01 (2)	59.57	0.00 (10)	309.80
sko100e	100	149,150	0.00 (6)	42.21	0.00 (9)	42.86	0.00 (7)	45.54	0.00 (10)	309.10
sko100f	100	149,036	0.03 (2)	46.19	0.00 (7)	59.36	0.00 (7)	52.76	0.00 (4)	310.30
wil100	100	273,038	0.02 (2)	82.89	0.00 (10)	27.40	0.00 (7)	63.95	0.00 (10)	316.60
tho150	150	8,133,398	0.14 (0)	106.87	0.04 (0)	120.29	0.07 (0)	48.30	0.01 (0)	1,991.70

Notes. The success rate of reaching the best known result over 10 executions is indicated between parentheses. Computing times are given in minutes (experimentally determined for BLS and BMA, from James et al. 2009 for CPTS).

into an unfavorable position when competing with fine-tuned, specially designed heuristics. For example, BLS does a single exchange of variable values (swap) in 0.00021 ms. RDSolver, on the other hand, requires 0.027 ms to do the same task. Thus, for the same computing time, BLS evaluates 100x more solution candidates than RDSolver does. Regardless, as our numerical experiments show, for most of the benchmark problems, RDSolver performance is comparable to the best QAP heuristics known at the time of writing this paper.

4.2. Hard Problems of Large Size

In this section, we turn our attention to a special class of instances that were constructed to be difficult for heuristic methods. In Drezner et al. (2005) the authors propose two new problem classes: dre* instances that are ill-conditioned for local search methods, and tai*xx*e*yy* instances that are difficult to solve by global search heuristic methods. RDSolver results and results for the BLS and BMA algorithms for both sets of problem instances are presented in the subsequent sections. Results for BLS and BMA are not reported in Benlic and Hao (2013 or 2015) for these problem instances. Therefore, to provide a fair comparison with RDSolver, we experimented with the BLS and BMA algorithms to find the best runtime settings for them. We found that setting the initial jump magnitude L_0 to $0.4n$ for tai*xx*e*yy* and to $0.15n$ for dre* instances produced the best results. Other runtime parameters from Table 1 in Benlic and Hao (2015) did not change.

4.2.1. Tai*xx*e*yy* Instances.

In Drezner et al. (2005) the authors propose seven groups of tai*xx*e*yy* problem instances,[10] with problem size ranging from 27 to 729 (tai27e*yy* to tai729e*yy*), that are difficult to solve for global search heuristic methods. Tai* problem instances were constructed with the following goals: (a) local optima (for a transposition neighborhood) are relatively far apart; (b) instances should model large problems (large n); (c) the difficulty of the instances must grow with size. RDSolver results for all seven groups of difficult problem instances are presented in Tables 3–5. Each instance was solved 10 times, with repetitions starting from randomly generated initial points. Table 3 shows the results for the tai27e*yy* and tai45e*yy* problem instances. All methods were able to find the BKS for all instances 10 out of 10 times, but RDSolver was significantly faster than BLS and BMA.

Table 4 shows the results for the tai75e*yy* problem instances. As in Table 2, the results were compared to the best known solution (BKS), and the average percentage difference $\bar{\delta}$ from the BKS for the 10 runs is presented along with the number of times out of 10 the

Table 3. Comparative Results Between RDSolver and the Two Best Performing QAP Approaches on tai27eyy and tai45eyy Instances

	tai27eyy, $n = 27$				tai45eyy, $n = 45$			
		RD	BLS	BMA		RD	BLS	BMA
yy	BKS	t [s]	t [s]	t [s]	BKS	t [s]	t [s]	t [s]
1	2,558	0.7	1.6	5.6	6,412	1.6	135.0	149.6
2	2,850	1.3	6.2	6.5	5,734	9.8	58.5	30.1
3	3,258	0.0	2.4	5.5	7,438	5.9	108.5	34.2
4	2,822	0.2	2.7	5.9	6,698	36.0	155.7	33.8
5	3,074	0.3	3.2	6.5	7,274	7.4	70.4	1,664.2
6	2,814	0.2	0.9	5.3	6,612	3.5	103.0	645.6
7	3,428	0.2	1.7	5.4	7,526	4.3	158.2	1,124.2
8	2,430	0.0	9.6	6.2	6,554	3.4	138.1	166.5
9	2,902	0.5	1.2	5.1	6,648	2.4	141.2	51.4
10	2,994	0.2	0.9	4.6	8,286	7.6	114.7	124.3
11	2,906	0.8	1.0	5.2	6,510	4.4	59.3	39.1
12	3,070	0.5	2.8	6.3	7,510	4.3	134.3	1,283.7
13	2,966	0.0	2.0	5.5	6,120	7.3	57.1	1,279.2
14	3,568	0.3	1.0	5.0	6,854	4.1	77.1	370.4
15	2,628	1.5	2.5	5.9	7,394	3.6	41.7	44.7
16	3,124	0.0	1.8	5.4	6,520	6.4	41.0	276.4
17	3,840	0.2	1.2	4.8	8,806	5.9	165.1	44.0
18	2,758	1.1	2.4	5.6	6,906	4.3	67.0	169.8
19	2,514	0.6	3.3	6.7	7,170	7.2	244.7	141.9
20	2,638	1.1	1.4	5.2	6,510	3.7	162.8	53.2

Notes. Each algorithm found the best known result in 10 of 10 executions. Computing times are given in seconds.

BKS was found. Both RDSolver and BMA significantly outperformed BLS on these instances, with RDSolver slightly outperforming BMA. RD also found a new BKS for tai75e06, marked in boldface characters, that we were later able to replicate using the BMA algorithm.

Table 5 shows the results for large taixxeyy instances. All of these instances were previously unsolved, except for the tai125e01 and tai343e01 results reported in Drezner et al. (2005). RDSolver found a new BKS for tai125e01 (marked in boldface characters), and a

Table 4. Comparative Results Between the RDSolver and the Two Best Performing QAP Approaches on tai75eyy Instances

		RD		BLS		BMA	
Name	BKS	$\bar{\delta}$	t [min]	$\bar{\delta}$	t [min]	$\bar{\delta}$	t [min]
tai75e01	14,488	0.00 (10)	8.0	6.87 (1)	67.5	0.38 (8)	36.2
tai75e02	14,444	2.44 (1)	40.3	6.25 (0)	44.9	0.21 (9)	43.2
tai75e03	14,154	0.00 (10)	3.4	1.51 (1)	48.0	0.00 (10)	14.7
tai75e04	13,694	0.00 (10)	7.2	2.37 (0)	67.7	0.00 (10)	21.6
tai75e05	12,884	0.00 (10)	12.6	1.55 (0)	43.4	0.00 (10)	11.0
tai75e06	**12,534**	0.00 (10)	5.4	2.75 (0)	52.1	0.00 (7)	12.4
tai75e07	13,782	0.06 (9)	36.1	7.14 (0)	67.6	0.68 (5)	72.9
tai75e08	13,948	0.00 (10)	12.6	8.02 (0)	60.8	0.00 (10)	33.2
tai75e09	12,650	0.00 (10)	4.1	3.54 (0)	66.6	0.00 (10)	13.2
tai75e10	14,192	0.00 (10)	2.4	5.50 (0)	54.7	0.33 (9)	23.2
tai75e11	15,250	0.00 (10)	5.9	4.83 (0)	54.9	0.66 (8)	30.6
tai75e12	12,760	1.46 (5)	60.7	6.96 (0)	55.2	0.37 (8)	41.3
tai75e13	13,024	0.00 (10)	7.2	3.97 (0)	40.9	0.00 (10)	11.9
tai75e14	12,604	0.00 (10)	6.1	3.17 (1)	61.5	0.00 (10)	27.6
tai75e15	14,294	0.00 (10)	5.1	3.05 (0)	71.1	0.00 (10)	24.0
tai75e16	14,204	0.00 (10)	5.2	1.91 (0)	59.9	0.00 (10)	16.0
tai75e17	13,210	0.00 (10)	5.5	2.98 (0)	78.1	0.00 (10)	14.3
tai75e18	13,500	0.00 (10)	6.3	6.89 (0)	50.2	0.00 (10)	22.6
tai75e19	12,060	0.00 (10)	4.3	2.46 (0)	63.8	0.08 (9)	25.8
tai75e20	15,260	0.00 (10)	2.4	1.77 (1)	68.1	0.00 (10)	13.1

Notes. The success rate of reaching the best known result over 10 executions is indicated between parentheses. Computing times are given in minutes. All problems of equal size, $n = 75$. Max execution time: 2 h.

Table 5. Results from RDSolver on Previously Unsolved Instances of taixxeyy

	tai125eyy $n = 125$		tai175eyy $n = 175$		tai343eyy $n = 343$		tai729eyy $n = 729$	
yy	BFS	t [min]	BFS	t [min]	BFS	t [min]	BFS	t [min]
1	**35,426**	166.4	57,540	181.6	145,862	1,026.8	469,650	1,187.3
2	36,202	123.8	51,036	216.2	154,018	746.5	475,184	1,187.5
3	30,498	157.2	53,900	165.8	144,278	977.8	447,854	1,155.6
4	33,084	149.0	63,182	163.9	162,092	755.3	455,268	1,157.6
5	38,432	109.8	51,278	167.8	14,2110	701.4	475,466	1,188.7
6	35,546	102.3	54,752	187.1	144,274	653.7	466,946	1,174.9
7	32,712	172.7	52,502	179.7	154,776	607.0	454,480	1,178.2
8	36,354	150.1	57,304	161.3	133,770	593.8	835,098	1,187.9
9	35,008	126.5	53,238	172.1	143,018	779.8	427,478	1,188.1
10	34,898	84.8	52,010	162.4	152,828	848.5	457,434	1,127.4
11	33,082	175.6	54,892	156.7	14,6,446	813.9		
12	32,326	145.0	59,564	171.7	162,954	888.7		
13	35,380	239.6	59,840	100.6	137,836	1,076.8		
14	30,460	123.6	55,520	198.0	150,428	852.4		
15	34,328	196.0	49,668	236.1	156,682	749.8		
16	32,674	147.6	55,968	158.7	154,264	1,077.8		
17	35,512	123.9	58,572	97.2	136,650	721.0		
18	38,702	119.7	51,714	123.4	136,694	886.8		
19	33,034	168.5	52,298	216.5	150,486	796.2		
20	31,988	143.0	56,616	206.8	151,552	785.5		

Notes. The best found solution value (BFS) found over 10 executions. Max execution time: 5 h for tai125* and tai175*, 10 h for tai343e*, and 20 h for tai729e* instances.

solution with a gap of 7% from the BKS of tai343e01 (which is 136,288). As BMA and BLS solutions for tai125eyy were much worse than of RDSolver (up to 60% worse) and took much longer to obtain, we decided not to run those algorithms on the larger instances.

4.2.2. dre* instances. The dre* problem instances[11] are specially designed to be hard for heuristics to solve because they have a very challenging solution landscape. The dre* problem instances are based on a rectangular grid where all nonadjacent nodes have zero weight, making the value of the objective function increase steeply with just a slight change from the optimal permutation. The neighborhood of the optimum consists of solutions with much higher objective function values surrounded by "flat" areas, which makes dre* instances very hard to solve because the likelihood of finding a global optimum "valley" falls rapidly with an increase in the problem size. The best known solutions for the dre* problems have been found by branch and bound in Drezner et al. (2005).

In Table 6, we report computational results obtained for different dre* instances. As in Table 2, the results are compared to the optimal solution (OPT), and the average percentage difference $\bar{\delta}$ from the OPT for the 10 runs is presented along with the number of times out of 10 the OPT was found. The results confirm dre* instances to be more difficult when compared with other instances of the QAPLIB of similar sizes. For dre* instances with $n \geq 56$, RDSolver could not find the optimal solution on every run, but got stuck in local optima with $\bar{\delta}$ up to 40%, which is in a sharp contrast with, for example, hard to solve sko* instances

Table 6. Comparative Results Between RDSolver and the Two Best Performing QAP Approaches on dre* instances

			RD		BLS		BMA	
Name	n	BKS	$\bar{\delta}$	t [min]	$\bar{\delta}$	t [min]	$\bar{\delta}$	t [min]
dre30	30	508	0 (10)	0.5	0 (10)	0.2	0 (10)	0.2
dre42	42	764	0 (10)	17.5	0 (10)	3.8	0 (10)	1.9
dre56	56	1,086	15.7 (3)	71.8	23.5 (3)	47.5	22.4 (1)	44.5
dre72	72	1,452	29.5 (0)	34.2	40.8 (0)	63.2	37.1 (0)	86.5
dre90	90	1,838	36.9 (0)	53.4	57.3 (0)	39.8	53.5 (0)	120
dre110	110	2,264	39.2 (0)	68.3	63.5 (0)	46.8	63.5 (0)	120
dre132	132	2,744	43.6 (0)	75.1	69.8 (0)	47.6	69.4 (0)	120

Note. Max execution time: 2 h.

from QAPLIB, for which $\bar{\delta}_! < 0.2\%$. The BLS and BMA results are even worse than RDSolver with the worst $\bar{\delta}$ approaching 70%. Note that all the algorithms often finish with poor results in less than the two-hour maximum time, indicating that they are stuck in a local optimum.

5. RDSolver Performance Investigation

Although RDSolver and recent heuristics had very good results for most of the test problems, there were certain problems for which RDSolver had the best results and other problems for which recent heuristics had the best results. We did some investigation into the general structure of the test problems to see if we could determine the types of problems for which RDSolver seems to best apply and to see if there are preprocessing steps that could improve our performance on certain problem types. The investigations were in two areas—the general nature of the problem structure, as described in Section 5.1, and the relationship of the solution to the Gilmore-Lawler lower bound, as described in Section 5.2.

5.1. Flow and Distance Graphs

In Benlic and Hao (2015) the fitness distance correlation from Jones and Forrest (1995) is computed for the taixxa, taixxb, and sko instances. In this computation, the distance between two solutions is defined as the number of locations that are assigned different facilities in each solution. The difference in fitness between two solutions is defined as the difference in solution objective value. Their analysis based on the fitness distance correlation indicates that the taixxa problems should be the most difficult to solve, while taixxb problems should be more difficult to solve than the sko problems. Of the benchmark problems, the taixxa problems were the most difficult for RDSolver, but taixxb were easier for RDSolver than sko, so there was not a perfect correlation with the fitness distance correlation analysis.

A different characterization of the problem difficulty is determined by treating the flow or distance matrix as a weighted adjacency matrix in a general graph and performing an analysis on the graph. We define the graph generated from the flow matrix as the *flow graph* and the graph generated from the distance matrix as the *distance graph*. We concentrated on the flow graph because most of the test problems had very dense and similar distance graphs. A relatively sparse flow graph can help partition the problem into groups of facilities that should all be located near each other. The values for the flow obviously matter also, and some of the easy problems in QAPlib have large values for some flows that make it easy to fix some of the facility locations. RDSolver subproblems result from reshuffling of a small number (usually two or three) of facility locations. A dense flow graph, especially with relatively uniform flow values, could mean that it is very difficult for the RD local search to make progress and that RDPerturb may also have to change large numbers of facility locations to make progress.

Table 7. RDSolver Performance vs. Flow Matrix Density

Instance or group	No. of instances	Average density (%)	Min density (%)	RDSolver performance vs. heuristics
taixxa	5	98	98	Worse
taixxb	5	43	24–33	Similar
tai45(75)e	40	53	51	Better
sko	9	68	56	Similar
lipa	4	99	99	Worse
wil100	1	89	82	Similar
tho150	1	63	46	Worse

Notes. Average(Min) density is the average(minimum) number of other facilities with which a facility has flow. RDSolver performance is relative to the BLS and BMA results reported in Section 4.

We did an analysis of the average and minimum node degree of the flow graph for the benchmarks we reported in Tables 2 and 4. The results are shown in Table 7, along with a qualitative comparison of RDSolver results to the heuristic results. The taixxa instances have flow graphs with high average degree because the flow is a uniform random variable between 0 and 100. The lipa instances in QAPlib, which are not considered hard and thus were not reported in Table 2, have a similarly dense structure. Although RDSolver can solve these problems, it takes much longer than the heuristics. The taixxb instances, for which RDSolver does about the same as the heuristics, and the taixxeyy instances, for which RDSolver does very well, both have a less dense flow graph. Looking at the entire table, it is clear there is not a perfect correlation with RDSolver performance and flow graph density, but in the future we could consider using the flow graph structure to adjust RDSolver parameters, especially if we seem to be stuck in a local optimum.

5.2. Challenging Instances

In Section 4.2.2 we reported results for the dre* problem instances. While RDSolver could successfully solve small problem instances, it became stuck in local optima with an average relative gap of up to 40% for instances with $n \geq 56$. To improve RDSolver performance on the dre* instances, we evaluated the Gilmore-Lawler bound (GLB), which is a well-known lower bound for the QAP.

The dre* instances have the useful property that, for each facility, there are at most four nonzero flows associated with that facility. More specifically, each facility corresponds to a particular grid point (i, j), and the nonzero flows for that facility are the grid points a unit distance away from (i, j). Therefore, if we simply place each facility at the grid point corresponding to (i,j), then all of its nonzero flows are routed along edges of length 1, which is certainly the best possible. The GLB linear assignment problem cannot improve upon this assignment, which is why the GLB is tight for dre* instances. A rigorous proof of this statement is provided in the online supplement. If we can construct a feasible solution that satisfies the bound, the solution is optimal. If the feasible solution construction is very fast, it can be run before starting RDSolver to possibly find a good initial feasible solution or even an optimal solution.

5.2.1. Gilmore-Lawler Based Heuristic. When computing the GLB, one would ideally like to determine if the bound is tight and if so, construct a feasible, optimal solution. Unfortunately, Li et al. (1994) shows that checking if the GLB bound is tight is an NP-complete problem. Instead of an exact method, we develop a construction heuristic to generate a potentially good feasible solution. Note that the GLB specifies a feasible solution for the QAP, so the most naive heuristic would be to simply use that solution. In certain cases, this may work well, but in cases where there are many zero flows between facilities, a very poor solution may be constructed. An alternative heuristic that provides a better solution is shown in Algorithm 4. The algorithm starts by initializing the set of already assigned facilities, S, with the location that has the fewest connections placed in its GLB location (lines 5–8). Then, a matching flow of the neighbors to the assigned facilities is constructed in line 13. If the matching is correct, then all the matched facilities are added to sets S and R (line 15). The procedure (lines 9–16) is repeated until all facilities are placed or a failure is detected, indicating that the problem is infeasible.

Algorithm 4 (GLB heuristic)

```
Require: Problem definition: distance matrix $A$
             and flow matrix $B$
         GLB solution definitions: location of facility $i$
           in the GLB solution; $g(i)$ cost of placing
         facility $i$ to location $j$, $c_{i,j}$; location of facility $j$
         in the subproblem used to define $c_{i,j}$, $f(i,j)$
Ensure: Initial feasible, possibly optimal solution
 1: $F :=$ {set of all facilities}; $L :=$ {set of all locations};
    S := {set of already assigned facilities}
 2: $R := \{(i,j): i \in F, j \in L\}$
 3: $C_i := \{k \in L: c_{i,g(i)} = c_{i,k}\}$
 4: $D_{i,j} := \{k \in L: b_{g(i),f(i,j)} = b_{g(i),k}\}$
 5: $N_i := \{j \in F: a \in A, a_{ij} > 0\}$
 6: $i^* \leftarrow \arg\min_{i\in F}(|N_i|)$
 7: $R \leftarrow \{(i^*, g(i^*))\}$
 8: $S \leftarrow \{i^*\}$
 9: while $|S| < |F|$
10:     $P := \{j \in F \setminus S: \exists s \in S, a_{sj} > 0\}$
11:     Construct a complete bipartite graph $G=(P,L,E)$
12:     For each $i \in P$, set edge $(i,k) \in E$ weight to be
           1, if $k \in C_i \cap (\bigcap_{j\in\{N_i\cap S\}} D_{i,j})$
           2, otherwise
13:     Find min weight matching of $G$
14:     Assign facilities to a randomly selected
          optimal matching[12]
15:     Update $S$ and $R$ with matched facilities and
          corresponding location assignments
          (found in the previous step)
16: end while
16: return $R$ (Locations for all facilities
      in a feasible solution).
```

The GLB heuristic runs in a few seconds, so it can be run multiple times with potentially different results due to the randomness in line 14. It successfully places facilities in all dre* instances correctly with probability 1/2 because, due to the grid structure of the dre* instances, a least connected corner node will be placed correctly, and then all other facilities will be placed at their correct GLB positions or at their transposed GLB positions. By running the heuristic 10 times (e.g.,) it is very likely to find the optimal solution. The tolerances for sets C_i and $D_{i,j}$ can be loosened (lines 3 and 4) to generate potentially good suboptimal solutions. Thus, a potential modification to RDSolver and other heuristics is to use the GLB heuristic to create an initial feasible solution, which may be optimal for some types of problem instances.

6. Conclusions

This paper has presented randomized decomposition (RD), a methodology specifically designed for hard, nonlinear, nonconvex mathematical programs. The solution technique employed by RD is to randomly partition a problem into subsets of decision variables, solve the subproblem of optimizing the variables in each subset with all other variables held at their current values, then repeat as described in Section 2. Specifically, for the QAP, we select a random subset of the matrix element assignment, construct all permutations of the assignment, select the permutation that provides the optimal objective function value, then repeat with another subset. RD differs from most other solution techniques in that it is a brute force technique that tries large numbers of solutions rather than attempting to find the best search direction or mutation. This characteristic makes it easy to parallelize by doing a local neighborhood search on each thread of a multithreaded computer, which we believe makes it perfectly aligned with the current trend toward massive multithreading.

RDSolver is a general purpose mathematical programming solver that combines RD local neighborhood search with a metaheuristic large neighborhood search method called RDPerturb to escape from local optima. We have applied RDSolver to over 400 QAP problems in QAPLIB, and specially constructed hard problems, thus illustrating the breadth of QAP problems to which RD applies. Despite being a general purpose approach, RD is competitive with heuristics targeted at QAP problems, and we have obtained two new best known solutions as well as providing solutions for 68 previously unsolved tai* QAP problems (Table 5). In addition, we have identified QAP problems for which RDSolver performs poorly and developed a fast heuristic based on the Gilmore-Lawler bound that could provide a good initial feasible solution or even the optimal solution in certain cases.

In addition to QAP problems, RDSolver has been successfully applied to a wide range of problems, as described in the online supplement and on our website, Oracle Labs (2016). It is currently shipping in Oracle products for project portfolio optimization and vehicle routing and is under evaluation for use in other areas. There are a number of enhancements to our RDSolver implementation that we are exploring to improve performance. There are multiple ways to parallelize RD, and we are experimenting with various options. Our code is in Java and has not been completely optimized; better optimization and/or converting to C/C++ could improve execution speed. It is possible that there are parameter settings, such as subproblem size, that could improve execution efficiency. Finally, open research questions include defining the problems that are most amenable to solution by RD and developing solution bounds.

Endnotes

[1] In practice runtime (rT) is the normal termination criterion. For the benchmark results provided herein, other termination criteria, such as reaching the best known solution or inability to improve the solution for a long time, may be used.

[2] Subproblems may be solved by any applicable method. Enumeration is often used with RD to solve the subproblems.

[3] Throughout this document we use set theory mathematical symbols to describe operations on sets (https://enwikipedia.org/wiki/List_of_mathematical_symbols_by_subject).

[4] Throughout this document, the uniform distribution is implied when the phrase "at random" or "selected randomly" is used without specifying a distribution.

[5] An inequality constraint $g_p(x) \geq 0$ is said to be *active* or *tight* if satisfied at a point $\bar{x}$ with $g_p(\bar{x}) = 0$. The equality constraint $h_i(x) = 0$ is always an active constraint at any point $\bar{x}$ satisfying it (Murty 1988).

[6] The set includes at least one feasible solution, which is x_{start}. Note that we ignore the remote possibility of choosing the same candidate multiple times because of the random nature of RD, Algorithm 2, which may find a different local optimum starting from the same $x_{\text{candidate}}$.

[7] http://www.info.univ-angers.fr/pub/hao/BLS_code.cpp.

[8] http://www.info.univ-angers.fr/pub/hao/BMA.html.

[9] http://www.seas.upenn.edu/qaplib/.

[10] http://mistic.heig-vd.ch/taillard/problemes.dir/qap.dir/qap.html.

[11] http://cbeweb-1.fullerton.edu/isds/zdrezner/programs.htm.

[12] We can randomly generate different optimal matchings by perturbing edge weights randomly (e.g., by adding ϵ small enough to generate different matchings of weight P).

References

Armour GC, Buffa ES (1963) A heuristic algorithm and simulation approach to relative location of facilities. *Management Sci.* 9(2):294–309.

Benlic U, Hao J (2013) Breakout local search for the quadratic assignment problem. *Appl. Math. Comput.* 219(9):4800–4815.

Benlic U, Hao JK (2015) Memetic search for the quadratic assignment problem. *Expert Systems Appl.* 42(1):584–595.

Burkard RE (2013) The quadratic assignment problem. Pardalos PM, Du DZ, Graham RL, eds. *Handbook of Combinatorial Optimization* (Springer, New York), 2741–2814.

Burkard RE, Karisch SE, Rendl F (1997) QAPLIB—A quadratic assignment problem library. *J. Global Optim.* 10(4):391–403. Revised 02.04.2003 (electronic update): http://www.seas.upenn.edu/qaplib/.

Cela E (1998) *The Quadratic Assignment Problem: Theory and Algorithms*, Combinatorial Optimization, Vol. 1 (Springer, Boston).

Drezner Z (2005) The extended concentric tabu for the quadratic assignment problem. *Eur. J. Oper. Res.* 160(2):416–422.

Drezner Z (2008a) Extensive experiments with hybrid genetic algorithms for the solution of the quadratic assignment problem. *Comput. Oper. Res.* 35(3):717–736.

Drezner Z (2008b) Tabu search and hybrid genetic algorithms for quadratic assignment problems. Jaziri W, ed. *Tabu Search* (InTech, London), 89–108.

Drezner Z, Hahn P, Taillard ÉD (2005) Recent advances for the quadratic assignment problem with special emphasis on instances that are difficult for meta-heuristic methods. *Ann. Oper. Res.* 139(1):65–94.

Helsgaun K (2000) An effective implementation of the Lin-Kernighan traveling salesman heuristic. *Eur. J. Oper. Res.* 126(1):106–130.

James T, Rego C, Glover F (2009) A cooperative parallel tabu search algorithm for the quadratic assignment problem. *Eur. J. Oper. Res.* 195(3):810–826.

Jones T, Forrest S (1995) Fitness distance correlation as a measure of problem difficulty for genetic algorithms. *Proc. 6th Internat. Conf. Genetic Algorithms* (Morgan Kaufmann, San Francisco), 184–192.

Koopmans TC, Beckmann M (1957) Assignment problems and the location of economic activities. *Econometrica* 25(1):53–76.

Kovac M (2013) Solving quadratic assignment problem in parallel using local search with simulated annealing elements. 2013 *Internat. Conf. Digital Tech. (DT), Zilina, Slovakia*, 18–20.

Li Y, Pardalos PM, Ramakrishnan KG, Resende MGCM (1994) Lower bounds for the quadratic assignment problem. *Ann. Oper. Res.* 50(1):387–410.

Loiola EM, de Abreu NMM, Boaventura-Netto PO, Hahn P, Querido T (2007) A survey for the quadratic assignment problem. *Eur. J. Oper. Res.* 176(2):657–690.

Murty KG (1988) *Linear Complementarity, Linear and Non Linear Programming*, Sigma Series in Applied Mathematics (Heldermann Verlag, Berlin).

Oracle Labs (2016) Randomized decomposition project. Accessed August 1, 2017, https://labs.oracle.com/pls/apex/f?p=labs:49:::::P49_PROJECT_ID:141.

Ramkumar A, Ponnambalam S, Jawahar N (2009) A population-based hybrid ant system for quadratic assignment formulations in facility layout design. *Internat. J. Adv. Manufacturing Tech.* 44(5–6):548–558.

Sahni S, Gonzalez T (1976) P-complete approximation problems. *J. ACM* 23(3):555–565.

Stutzle T (2006) Iterated local search for the quadratic assignment problem. *Eur. J. Oper. Res.* 174(3):1519–1539.