INFORMS Journal on Computing

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Very Large-Scale Neighborhood Search for the Quadratic Assignment Problem

Ravindra K. Ahuja, Krishna C. Jha, James B. Orlin, Dushyant Sharma,

INFORMS Journal on Computing
Vol. 19, No. 4, Fall 2007, pp. 646–657
ISSN 1091-9856 | EISSN 1526-5528 | 07 | 1904 | 0646

DOI 10.1287/ijoc.1060.0201
© 2007 INFORMS

Very Large-Scale Neighborhood Search for the Quadratic Assignment Problem

Ravindra K. Ahuja
Department of Industrial and Systems Engineering, University of Florida,
Gainesville, Florida 32611, ahuja@ufl.edu

Krishna C. Jha
Innovative Scheduling, Inc., Gainesville Technology Enterprise Center,
Gainesville, Florida 32641, krishna@innovativescheduling.com

James B. Orlin
Sloan School of Management, Massachusetts Institute of Technology,
Cambridge, Massachusetts 02139, jorlin@mit.edu

Dushyant Sharma
Department of Industrial and Operations Engineering, University of Michigan,
Ann Arbor, Michigan 48109, dushyant@umich.edu

The quadratic assignment problem (QAP) consists of assigning n facilities to n locations to minimize the total weighted cost of interactions between facilities. The QAP arises in many diverse settings, is known to be NP-hard, and can be solved to optimality only for fairly small instances (typically, $n \leq 30$). Neighborhood search algorithms are the most popular heuristic algorithms for solving larger instances of the QAP. The most extensively applied neighborhood structure for the QAP is the 2-exchange neighborhood. This neighborhood is obtained by swapping the locations of two facilities; its size is therefore $O(n^2)$. Previous efforts to explore larger neighborhoods (such as 3-exchange or 4-exchange neighborhoods) were not very successful, as it took too long to evaluate the larger set of neighbors. In this paper, we propose very large-scale neighborhood (VLSN) search algorithms when the size of the neighborhood is very large, and we propose a novel search procedure to enumerate good neighbors heuristically. Our search procedure relies on the concept of an improvement graph that allows us to evaluate neighbors much faster than existing methods. In this paper, we present extensive computational results of our algorithms when applied to standard benchmark instances.

Key words: neighborhood search; heuristics; combinatorial optimization; quadratic assignment problem; computational testing
History: Accepted by Michel Gendreau, Area Editor for Heuristic Search and Learning; received July 2002; revised August 2003, March 2005; accepted June 2006. Published online in *Articles in Advance* August 31, 2007.

1. Introduction

The quadratic assignment problem (QAP) is a classical combinatorial optimization problem widely regarded as one of the most difficult in its class. Given a set $N = \{1, 2, \ldots, n\}$, and $n \times n$ matrices $F = \{f_{ij}\}$, and $D = \{d_{ij}\}$, the QAP is to find a permutation of the set N that minimizes

$$z(\phi) = \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{\phi(i)\phi(j)}. \tag{1}$$

The function $z(\phi)$ is the sum of n^2 flow costs between n facilities. (The term $f_{ij} d_{\phi(i)\phi(j)}$ represents the cost of flow from facility i to facility j.) Therefore, the QAP's aim is to determine an assignment of facilities to locations to minimize the total cost of flow between the facilities. The matrices F and D are typically symmetric but are not required to be so. In our algorithms, we allow for asymmetric instances so do not assume that $f_{ij} = f_{ji}$ or $d_{ij} = d_{ji}$.

The QAP arises as a natural problem in facility layout. In this context, the set N represents a set of n facilities (numbered 1 through n) that need to be assigned to locations (numbered 1 through n). The matrix $F = \{f_{ij}\}$ represents the flow between different facilities, and the matrix $D = \{d_{ij}\}$ represents the distance between locations. For example, if the facilities are departments in a university, then the flow f_{ij} could be the average number of people walking daily from department i to department j. The decision variable $\phi(i)$, $1 \leq i \leq n$ represents the location assigned to facility i. Since there are n facilities and n locations and a facility can be assigned to exactly one location, there is a one-to-one correspondence between the feasible solutions of QAP and the permutations ϕ.

In addition to facility layout, the QAP arises in many other applications, such as allocation of plants to candidate locations, backboard wiring problems, design of control panels and typewriter keyboards, turbine balancing, and ordering of interrelated data on a magnetic tape; see Malucelli (1993), Pardalos et al. (1994), Burkard et al. (1998), and Çela (1998). Given the wide range of applications and the difficulty of solving the problem, the QAP has been investigated extensively. The QAP is known to be NP-hard, and a variety of exact and heuristic algorithms have been proposed. Exact algorithms for solving QAP include approaches based on (i) dynamic programming (Christofides and Benavent 1989); (ii) cutting planes (Bazaraa and Sherali 1980); and (iii) branch-and-bound algorithms (Lawler 1963, Pardalos and Crouse 1989, Hahn et al. 2001, Anstreicher et al. 2002). Among these, the branch-and-bound algorithms are the most successful, but they are generally unable to solve moderately large problems.

Since applications of the QAP often give rise to moderately large problems, there is a need for effective heuristics that can solve larger problems. A wide variety of heuristic approaches have been developed for the QAP. These can be classified into the following categories: (i) *construction methods* (Buffa et al. 1964, Müller-Merbach 1970); (ii) *limited enumeration methods* (West 1983, Burkard and Bönniger 1983); (iii) *GRASP* (greedy randomized adaptive search procedure) (Li et al. 1994); (iv) *simulated-annealing methods* (Wilhelm and Ward 1987); (v) *tabu-search methods* (Skorin-Kapov 1990, Taillard 1991); (vi) *genetic algorithms* (Fleurent and Ferland 1994, Tate and Smith 1995, Ahuja et al. 2000, Drezner 2003); and (vii) *ant systems* (Maniezzo et al. 1994). The tabu-search method of Taillard (1991), the GRASP method of Li et al. (1994), and the genetic algorithm of Drezner (2003) yield the best results for QAPLIB instances.

Burkard et al. (1998) observe that the current neighborhood search (or tabu-search and simulated-annealing) algorithms for the QAP use the 2-*exchange* neighborhood structure. A permutation ϕ' is called a 2-*exchange neighbor* of the permutation ϕ if it can be obtained from ϕ by switching the values of two entries. Clearly, the number of 2-exchange neighbors of a permutation is $O(n^2)$. There has been very limited effort to explore larger neighborhood structures for the QAP, as the time required to identify an improved neighbor is too high. We investigate the neighborhood structure based on *multi-exchange*, which is a natural generalization of 2-exchanges. A multi-exchange is specified by a cyclic sequence $C = i_1 - i_2 - \cdots - i_k - i_1$ of facilities such that $i_p \neq i_q$ for $p \neq q$. This multi-exchange implies that facility i_1 is assigned to the location $\phi(i_2)$, facility i_2 to $\phi(i_3), \ldots,$ and facility i_k is assigned to $\phi(i_1)$. The locations of all other facilities do not change. If the objective function improves after a multi-exchange, we call it a *profitable multi-exchange*. Let ϕ^C be the permutation obtained by applying the multi-exchange C to the permutation ϕ. In other words,

$$\begin{aligned} &\phi^C(i) = \phi(i) \quad \text{for } i \in N\backslash\{i_1, \ldots, i_k\}.\\ &\phi^C(i_p) = \phi(i_{p+1}) \quad \text{for } p = 1, \ldots, k-1, \quad \text{and} \qquad (2)\\ &\phi^C(i_k) = \phi(i_1). \end{aligned}$$

We define the length of a multi-exchange as the number of facilities involved in the corresponding cyclic sequence. In cyclic sequence $C = i_1 - i_2 - \cdots - i_k - i_1$, k facilities are involved so we call this cycle exchange a *multi-exchange of length k* or a *k-exchange*. Figure 1 illustrates an example of a 3-exchange. We note that a k-exchange can be generated by k different cyclic sequences. For example, the 3-exchange shown in Figure 1 can be generated by any of the sequences 3-7-6-3, 7-6-3-7, and 6-3-7-6.

Given a positive integer $2 \leq K \leq n$, the *K-exchange neighborhood structure* consists of all the neighbors of a permutation obtained by using multi-exchanges of length of no more than K. We note that the 2-exchange neighborhood structure is contained in the K-exchange neighborhood structure. The number of neighbors in the K-exchange neighborhood structure is $\Omega(\binom{n}{K}(K-1)!)$. This is very large even for moderate values of K. For example, if $n = 100$ and $K = 10$, then the K-exchange neighborhood may contain as many

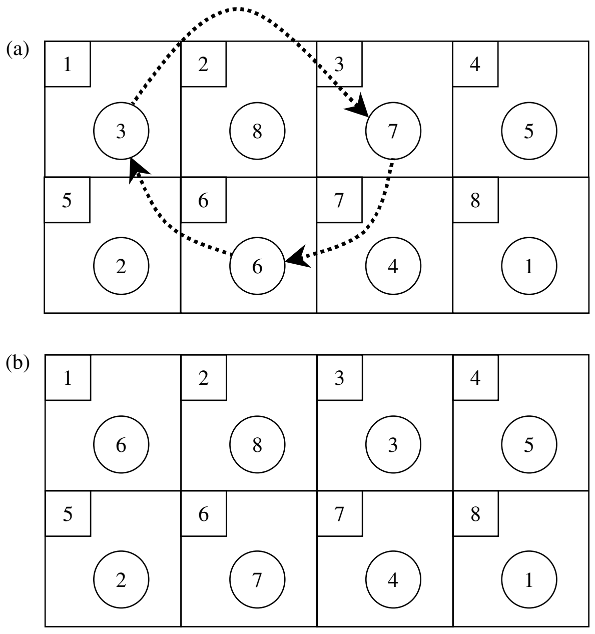

Figure 1 **(a) The Initial Assignment of Facilities to Locations and (b) The Assignment After the Cyclic Exchange 3-7-6-3**

as 6×10^{18} neighbors. This neighborhood structure falls under the category of *very large-scale neighborhood* (*VLSN*) *structure*, in which the size of the neighborhood is too large to be searched explicitly and we use implicit enumeration methods to identify improved neighbors.

Algorithms based on very large-scale neighborhood structures have been successfully used in the context of several combinatorial optimization problems; see Ahuja et al. (2002) and Deineko and Woeginger (2000) for surveys. One of the tools used in performing searches of very large-scale neighborhood structures is the *improvement graph*. In this technique, we associate a graph $G(\phi)$, called the improvement graph, with each feasible solution ϕ of the combinatorial optimization problem. The improvement graph $G(\phi)$ is constructed such that there is a one-to-one correspondence between every neighbor of ϕ and a directed cycle (possibly satisfying certain constraints) in the improvement graph $G(\phi)$. We also define arc costs in the improvement graph so that the difference in the objective-function value of a neighboring solution and the solution ϕ is equal to the cost of the constrained cycle corresponding to the neighbor. This transforms the problem of finding an improved neighbor into the problem of finding a negative-cost-constrained cycle in the improvement graph (assuming that the combinatorial optimization problem is a minimization). The concept of an improvement graph for partitioning problems was first proposed by Thompson and Orlin (1989) where a set of elements is partitioned into several subsets of elements to minimize the sum of the objective functions of the subsets. This technique has been used to develop several VLSN search algorithms for specific partitioning problems, such as the capacitated minimum spanning tree problem (Ahuja et al. 2001a, b) and the vehicle-routing problem. Thompson and Psaraftis (1993) and Gendreau et al. (1998) have used this neighborhood structure to solve vehicle-routing problems and obtained impressive results. The concept of multi-exchanges has also been studied by Glover (1996) who refers to it as *ejection chains* (see also Rego and Roucairol 1996, Kelly and Xu 1999).

The concept of the improvement graph was also used by Talluri (1996) and Ahuja et al. (2007) to search very large-scale neighborhoods for fleet-assignment problems arising in airline scheduling. Ergun (2001) proposed several improvement graphs for the vehicle-routing problem and for machine-scheduling problems.

We study improvement graphs for the multi-exchange neighborhood structure for the QAP. However, our application of the improvement graph is different. In previous applications, the improvement graph satisfied the property that the cost of the multi-exchange was equal to the cost of the corresponding (constrained) cycle in the improvement graph. This property is not ensured for the improvement graph for the QAP. Rather, the cost of the cycle is a very good approximation of the cost of the multi-exchange, and it allows us to enumerate good neighbors quickly. The improvement graph also allows us to evaluate the cost of a neighbor faster than with a normal method. Typically, evaluating a k-exchange neighbor for the QAP takes $O(nk)$ time, but with the improvement graph, we can do it in $O(k)$ time, on average.

We developed a generic search procedure to enumerate neighbors using improvement graphs and several implementations of it that enumerate the neighborhoods exactly as well as heuristically. We present a detailed computational investigation of local improvement algorithms based on our neighborhood search structures. We find that (i) locally optimal solutions obtained using multi-exchange neighborhood search algorithms are superior to those obtained using 2-exchange neighborhood search algorithms; (ii) generally, increasing the size of the neighborhood structure improves the quality of local optimal solutions, but after a certain point there are diminishing returns; and (iii) enumerating a restricted subset of neighbors is much faster than enumerating the entire neighborhood and can develop improvements that are almost as good.

In Section 2, we describe the improvement graph data structure for the QAP. In Section 3, we present a generic heuristic search procedure for the K-exchange neighborhood structure for the QAP. Section 4 describes several specific implementations of the generic search procedure. In Section 5, we describe the neighborhood search algorithm based on the generic search procedure. Section 6 describes an acceleration technique that speeds up the performance of the algorithm. We provide and analyze the computational results of our implementations in Section 7, and Section 8 summarizes our contributions.

2. Improvement Graph

One of our main contributions is development of the improvement graph to enumerate multi-exchanges for the QAP. In this section, we describe how to construct the improvement graph and how it may help us in evaluating multi-exchanges quickly. We require some network notation, such as cycles and paths and use the graph notation described by Ahuja et al. (1993).

Given a permutation ϕ and a k-exchange C, we denote the *cost* of the cyclic exchange by $Cost(\phi, C)$. This cost term represents the difference between the objective function values of ϕ^C and ϕ; in other words,

$$\begin{aligned} Cost(\phi, C) &= z(\phi^C) - z(\phi) \\ &= \sum_{i\in C}\sum_{j=1}^{n} f_{ij}(d_{\phi^C(i)\phi^C(j)} - d_{\phi(i)\phi(j)}) \\ &\quad + \sum_{i\in C}\sum_{\substack{j=1\\ j\neq i}}^{n} f_{ji}(d_{\phi^C(j)\phi^C(i)} - d_{\phi(j)\phi(i)}). \end{aligned} \tag{3}$$

Clearly, the cost of the k-exchange C can be computed in time $O(kn)$. We will demonstrate that by using improvement graphs, the cost of C can be computed in $O(k^2)$ time. This time can be further reduced to an average of $O(k)$ time. Since we choose k to be much smaller than n, the improvement graph allows us to evaluate a multi-exchange much faster than do standard methods. In fact, it also leads to dramatic improvements in the running time needed to identify traditional 2-exchanges.

We associate an improvement graph $G(\phi) = (N, A)$ with ϕ, which is a directed graph consisting of the node set N and arc set A. The node set N contains a node i for every facility i, and the arc set A contains an arc (i, j) for every ordered pair of nodes i and j in N. Each multi-exchange $C = i_1 - i_2 - \cdots - i_k - i_1$ defines a (directed) cycle $i_1 - i_2 - \cdots - i_k - i_1$ in $G(\phi)$ and, similarly, each (directed) cycle $i_1 - i_2 - \cdots - i_k - i_1$ in $G(\phi)$ defines a multi-exchange $i_1 - i_2 - \cdots - i_k - i_1$ with respect to ϕ. Therefore, there is a one-to-one correspondence between multi-exchanges with respect to ϕ and cycles in $G(\phi)$. We use C to denote both a multi-exchange and a cycle in $G(\phi)$; its type will be apparent from context. An arc $(i, j) \in A$ signifies that the facility i moves from its current location to the current location of facility j. In view of this interpretation, a cycle $C = i_1 - i_2 - \cdots - i_k - i_1$ signifies the following changes: facility i_1 moves from its current location to the location of facility i_2, facility i_2 moves from its current location to the location of facility i_3, and so on. Finally, facility i_k moves from its current location to the location of facility i_1.

We now associate a cost c_{ij}^{ϕ} with each arc $(i, j) \in A$. Ideally, we would like to define arc costs so that the cost of the multi-exchange C with respect to the permutation ϕ is equal to the cost of cycle C in $G(\phi)$. However, such a possibility would imply that P = Co-NP because one could efficiently determine if a solution was optimal. We will, instead, define arc costs so that the cost of the multi-exchange is "close" to the cost of the corresponding cycle. We define c_{ij}^{ϕ} as follows: the change in the cost of the solution ϕ when facility i moves from its current location to the location of facility j and all the other facilities do not move. Observe that this change indicates that after it there is no facility at location $\phi(i)$, and the location $\phi(j)$ has two facilities. Therefore, to determine the cost of the change, we need to take the difference of the costs of interactions between facility i and the other facilities before and after the change. Let ϕ' denote the function after the change. Then, $\phi'(l) = \phi(l)$ for $l \neq i$ and $\phi'(i) = \phi(j)$. Note that ϕ' is not a permutation because $\phi'(i) = \phi'(j)$. We define $c_{ij}^{\phi} = z(\phi') - z(\phi)$. Therefore,

$$\begin{aligned} c_{ij}^{\phi} &= z(\phi') - z(\phi) \\ &= \sum_{l=1}^{n} f_{il}(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)}) \\ &\quad + \sum_{\substack{l=1\\ l\neq i}}^{n} f_{li}(d_{\phi(l)\phi(j)} - d_{\phi(l)\phi(i)}), \end{aligned} \tag{4}$$

which captures the change in the cost of interaction *from* facility i to the other facilities.

The manner in which we define arc costs in the improvement graph does not ensure that the cost of the cycle C in $G(\phi)$, given by $\sum_{(i,j)\in C} c_{ij}^{\phi}$, will equal $Cost(\phi, C)$. The discrepancy in these two cost terms arises because when defining the arc cost c_{ij}^{ϕ}, we assume that the facility i moves from its current location to the location of facility j, but no other facilities move. However, in the multi-exchange C, several facilities do move, and so we do not correctly account for the cost of flow between facilities in C. We do, though, correctly account for the cost of flow between any two facilities if one of the two facilities is not in C. Next, we demonstrate that the cost term $Cost(\phi, C)$ can be computed by adding a corrective term to $\sum_{(i,j)\in C} c_{ij}^{\phi}$:

$$\begin{aligned} Cost(\phi, C) &= z(\phi^C) - z(\phi) \\ &= \sum_{i\in C}\sum_{j=1}^{n} f_{ij}(d_{\phi^C(i)\phi^C(j)} - d_{\phi(i)\phi(j)}) \\ &\quad + \sum_{i\in C}\sum_{\substack{j=1\\ j\neq i}}^{n} f_{ji}(d_{\phi^C(j)\phi^C(i)} - d_{\phi(j)\phi(i)}) \\ &= \sum_{i\in C}\sum_{\substack{j=1\\ j\notin C}}^{n} f_{ij}(d_{\phi^C(i)\phi(j)} - d_{\phi(i)\phi(j)}) \\ &\quad + \sum_{i\in C}\sum_{\substack{j=1\\ j\notin C}}^{n} f_{ji}(d_{\phi(j)\phi^C(i)} - d_{\phi(j)\phi(i)}) \\ &\quad + \sum_{i\in C}\sum_{j\in C} f_{ij}(d_{\phi^C(i)\phi^C(j)} - d_{\phi(i)\phi(j)}) \\ &\quad + \sum_{i\in C}\sum_{\substack{j\in C\\ j\neq i}} f_{ji}(d_{\phi^C(j)\phi^C(i)} - d_{\phi(j)\phi(i)}) \end{aligned}$$

$$
\begin{aligned}
&= \sum_{(i,j)\in C} c_{ij}^{\phi} - \sum_{i\in C}\sum_{j\in C} f_{ij}(d_{\phi^C(i)\phi(j)} - d_{\phi(i)\phi(j)}) \\
&\quad - \sum_{i\in C}\sum_{\substack{j\in C\\ j\neq i}} f_{ji}(d_{\phi(j)\phi^C(i)} - d_{\phi(j)\phi(i)}) \\
&\quad + \sum_{i\in C}\sum_{j\in C} f_{ij}(d_{\phi^C(i)\phi^C(j)} - d_{\phi(i)\phi(j)}) \\
&\quad + \sum_{i\in C}\sum_{\substack{j\in C\\ j\neq i}} f_{ji}(d_{\phi^C(j)\phi^C(i)} - d_{\phi(j)\phi(i)}) \\
&= \sum_{(i,j)\in C} c_{ij}^{\phi} + \sum_{i\in C}\sum_{j\in C} f_{ij}(d_{\phi^C(i)\phi^C(j)} - d_{\phi^C(i)\phi(j)}) \\
&\quad + \sum_{i\in C}\sum_{\substack{j\in C\\ j\neq i}} f_{ji}(d_{\phi^C(j)\phi^C(i)} - d_{\phi(j)\phi^C(i)}). \qquad (5)
\end{aligned}
$$

Equation (5) reveals that we can determine the cost of a multi-exchange C by first determining the cost of the cycle C in $G(\phi)$, which is $\sum_{(i,j)\in C} c_{ij}^{\phi}$, and then correcting it using the second and third term given in (5). This corrective term can be computed in time $O(k^2)$ if C is a k-exchange.

The improvement graph is useful. First, it allows us to determine the approximate cost of a multi-exchange C quickly. The cost term $\sum_{(i,j)\in C} c_{ij}^{\phi}$ is a reasonable estimate of the cost of the multi-exchange C. To see this, observe from (5) that the corrective term $Cost(\phi, C) - \sum_{(i,j)\in C} c_{ij}^{\phi}$ contains $O(k^2)$ interactions between facilities. However, n facilities have $O(n^2)$ interactions between them. If we choose k to be a relatively small fraction of n, then the corrective term (on average) will be substantially smaller than the total cost, and the cost of the cycle C in $G(\phi)$ will be a solid estimate of the cost of the multi-exchange C. For example, if $n = 100$ and $k = 5$, then there are 9,900 interactions between facilities and only 20 of them are counted incorrectly. If we use $k = 10$, then 90 of them are counted incorrectly, which is only 1% of the total interactions between facilities. Therefore, the improvement graph allows us to enumerate an extremely large set of neighbors quickly by simply summing the arc costs in a cycle.

The improvement graph also allows us to determine the correct cost of a multi-exchange faster than it normally takes to compute its cost. Normally, to compute the cost of a multi-exchange requires $O(kn)$ time, as we would need to update the cost interactions between k facilities (that move) with other facilities. However, with (5), we can compute the cost of a multi-exchange in $O(k^2)$ time. For example, if $n = 100$ and $k = 10$, then we can compute the cost of a multi-exchange about 10 times faster, which can make a substantial difference in an algorithm's performance.

The benefits that we derive from the use of the improvement graph come at a cost; we need to construct the improvement graph and calculate arc costs. It follows from (4) that we can construct the improvement graph from scratch in $O(n^3)$ time. But we need to compute the improvement graph from scratch just once. In all subsequent steps, we only *update* the improvement graph as we perform multi-exchanges. We illustrate in the next lemma that updating the improvement graph following a k-exchange takes only $O(kn^2)$ time. In Section 7, we will also demonstrate that our neighborhood search algorithms need to use k (4 and 5) only as on the benchmark tests higher values do not add much value. Therefore, it takes $O(n^2)$ time to update the improvement graph, which is quite efficient in practice. The time needed to construct and update the improvement graph, consequently, is relatively small, and it is well justified by the savings that we obtain in enumerating and evaluating multi-exchanges.

Lemma 1. *Given the improvement graph $G(\phi)$ and a k-exchange C with respect to ϕ, the improvement graph $G(\phi^C)$ can be constructed in $O(kn^2)$ time.*

Proof. The improvement graphs $G(\phi)$ and $G(\phi^C)$ have the same set of nodes and arcs. They differ only in arc costs. Each arc $(i, j) \in G(\phi^C)$ is one of the following two types: (i) either $i \in C$ or $j \in C$, or (ii) $i \notin C$ and $j \notin C$. There are $O(nk)$ arcs of type (i) and $O(n^2)$ arcs of type (ii). With (4), we can determine the cost of a type (i) arc in $O(n)$ time, thus requiring a total time of $O(n^2k)$ to compute the cost of all type (i) arcs. Next, we demonstrate that we can determine the cost of a type (ii) arc in $O(k)$ time, which also yields a total time of $O(n^2k)$ to compute the costs of all type (ii) arcs:

$$
\begin{aligned}
c_{ij}^{\phi^C} &= \sum_{l=1}^{n} f_{il}(d_{\phi(j)\phi^C(l)} - d_{\phi(i)\phi^C(l)}) \\
&\quad + \sum_{\substack{l=1\\ l\neq i}}^{n} f_{li}(d_{\phi^C(l)\phi(j)} - d_{\phi^C(l)\phi(i)}) \\
&= \sum_{l\notin C} f_{il}(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)}) + \sum_{l\in C} f_{il}(d_{\phi(j)\phi^C(l)} - d_{\phi(i)\phi^C(l)}) \\
&\quad + \sum_{\substack{l\notin C\\ l\neq i}} f_{li}(d_{\phi(l)\phi(j)} - d_{\phi(l)\phi(i)}) \\
&\quad + \sum_{\substack{l\in C\\ l\neq i}} f_{li}(d_{\phi^C(l)\phi(j)} - d_{\phi^C(l)\phi(i)}) \\
&= \sum_{l=1}^{n} f_{il}(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)}) + \sum_{\substack{l=1\\ l\neq i}}^{n} f_{li}(d_{\phi(l)\phi(j)} - d_{\phi(l)\phi(i)}) \\
&\quad - \sum_{l\in C} f_{il}(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)}) - \sum_{\substack{l\in C\\ l\neq i}} f_{li}(d_{\phi(l)\phi(j)} - d_{\phi(l)\phi(i)}) \\
&\quad + \sum_{l\in C} f_{il}(d_{\phi(j)\phi^C(l)} - d_{\phi(i)\phi^C(l)})
\end{aligned}
$$

$$\begin{aligned}&+\sum_{\substack{l\in C\\ l\neq i}} f_{li}(d_{\phi^C(l)\phi(j)}-d_{\phi^C(l)\phi(i)})\\ &= c_{ij}^{\phi}-\sum_{l\in C} f_{il}(d_{\phi(j)\phi(l)}-d_{\phi(i)\phi(l)})\\ &\quad-\sum_{\substack{l\in C\\ l\neq i}} f_{li}(d_{\phi(l)\phi(j)}-d_{\phi(l)\phi(i)})\\ &\quad+\sum_{l\in C} f_{il}(d_{\phi(j)\phi^C(l)}-d_{\phi(i)\phi^C(l)})\\ &\quad+\sum_{\substack{l\in C\\ l\neq i}} f_{li}(d_{\phi^C(l)\phi(j)}-d_{\phi^C(l)\phi(i)}). \qquad (6)\end{aligned}$$

Since we already know that c_{ij}^{ϕ} and C are k-exchanges, we can evaluate (6) in $O(k)$ time, which establishes the lemma. □

3. Identifying Profitable Multi-Exchanges

Our algorithm for the QAP is a neighborhood search algorithm and proceeds by performing profitable multi-exchanges. To manage the number of multi-exchanges enumerated, we first enumerate 2-exchanges, followed by 3-exchanges, and so on, until we reach a specified value of k, denoted by K, which is the largest size of the multi-exchanges that we wish to perform. This enumeration scheme is motivated by the consideration that we look for larger multi-exchanges when smaller multi-exchanges cannot be found. In this section, we describe a generic search procedure for enumerating and identifying multi-exchanges using improvement graphs.

Our method for enumerating multi-exchanges with respect to a solution ϕ proceeds by enumerating directed paths of increasing lengths in the improvement graph $G(\phi)$, where the length of a path is the number of nodes in the path. Observe that each path $P = i_1 - i_2 - \cdots - i_k$ in the improvement graph has a corresponding cycle in the improvement graph $i_1 - i_2 - \cdots - i_k - i_1$ obtained by joining the last node of the path with the first node in the path; this cycle also defines a multi-exchange with respect to ϕ. Let $C(P)$ denote the multi-exchange defined by the path P. Our method for enumerating cycles of increasing lengths performs the following three steps repeatedly for increasing values of k, starting with $k = 2$. Let S^k denote a set of paths of length k in $G(\phi)$. We start with $S^1 = \{1, 2, \ldots, n\}$, which is the set of n paths of length 1, each consisting of a singleton node.

3.1. Path Extension

We consider each path $P \in S^{k-1}$ one by one and extend it by adding one node to it. To extend a path $P = i_1 - i_2 - \cdots - i_{k-1}$, we add the arc (i_{k-1}, i_k) for each $i_k \in N\backslash\{i_1, i_2, \ldots, i_k\}$ and obtain several paths of length k. Let $E(P)$ denote the set of all paths obtained by extending the path P. Additionally, let $P^k = \bigcup_{P\in S^{k-1}} E(P)$.

3.2. Cycle Evaluation

Each path $P \in P^k$ yields a corresponding multi-exchange $C(P)$. We evaluate each of these multi-exchanges and determine whether any of them is a profitable multi-exchange. If so, we return the best multi-exchange and stop; otherwise, we proceed.

3.3. Path Pruning

In this step, we prune several paths in the set P^k that are less likely to lead to profitable multi-exchanges. We call a procedure, *PathSelect*(P^k), that takes as input the set of paths P^k enumerated in the previous step and selects a subset S^k of it. This subset of paths is extended further in the next iteration for the next higher value of k. We describe in Section 4 several ways to implement the *PathSelect* procedure. Path pruning is critical in order to manage the number of paths enumerated.

Figure 2 summarizes the steps of our heuristic search procedure, which we call the *K-exchange search* procedure. Observe that in this procedure, K is a parameter and can be specified by the user. Increasing K may in general improve the quality of local optimal solutions obtained, but our computational investigations demonstrate that there are diminishing returns after $K = 4$; therefore, $K = 4$ is an effective value to use in the search procedure. For another implementation (Implementation 4) of *PathSelect*, as discussed in Section 4, we retain the value of $K = 5$. Also observe that the algorithm terminates in two ways: C^* is empty, or C^* is nonempty. If C^* is empty, then it implies that the algorithm has failed to find a profitable multi-exchange and the current solution ϕ

```
procedure K-exchange search;
begin
  $k \leftarrow 1$;
  let $S^1 \leftarrow N$ be the set of paths of length 1;
  $C^* \leftarrow \phi$ and $W^* \leftarrow 0$;
  while $S^k$ is nonempty and $k < K$ and $W^* \geq 0$ do
  begin
    $k \leftarrow k+1$;
    $P^k \leftarrow \bigcup_{P\in S^{k-1}} E(P)$;
    let $P_{\min} \in P^k$ be the path such that
      $Cost(\phi, C(P_{\min})) = \min\{Cost(\phi, C(P)) : P \in P^k\}$;
    if $W^* > Cost(\phi, C(P_{\min}))$ then
      $W^* \leftarrow Cost(\phi, C(P_{\min}))$ and $C^* \leftarrow C(P_{\min})$;
    $S^k \leftarrow PathSelect(P^k)$;
  end;
  return $C^*$;
end.
```

Figure 2 The Generic Search Procedure for Identifying Profitable Multi-Exchanges

is locally optimal. If C^* is nonempty, then the algorithm found a profitable multi-exchange C^*.

We will now analyze the complexity of the algorithm. Let p denote the maximum number of paths in any S^k. The *while* loop executes at most K times. In each execution of the while loop, it takes $O(pn)$ time to compute the set P^k, and it may contain as many as pn paths. Since computing the cost of a k-exchange for each $P \in P^k$ takes $O(k^2)$ time, we require $O(k^2pn)$ time to find a profitable k-exchange, if any. In Section 4, we will illustrate that the subroutine *PathSelect* takes $O(pn\log(pn))$ time. Since for most situations considered by us $\log(pn) < k^2$, the running time of the algorithm is $O(k^2pn)$.

Clearly, if we ignore the time taken by the procedure *PathSelect*, then the bottleneck operation in the generic search procedure evaluates the cost $Cost(\phi, C(P))$ of each path $P \in P^k$. Since $C(P)$ is a k-exchange with respect to the solution ϕ, with (5), we can determine its cost in $O(k^2)$ time. Next, we show that we can determine the cost of k-exchange $C(P)$ in $O(k)$ time.

The generic search procedure proceeds by enumerating paths in $G(\phi)$. Each path $P = i_1 - i_2 - \cdots - i_k$ in $G(\phi)$ defines a "path exchange" with respect to the solution ϕ in an obvious manner, which is the same as the k-exchange $C = i_1 - i_2 - \cdots - i_k - i_1$ except that we do not perform the last move of shifting facility i_k to the location of facility i_1. Alternatively, $\phi^P(i_l) = \phi(i_{l+1})$ for all $l = 1, 2, \ldots, k-1$, and $\phi^P(i) = \phi(i)$ for all $i \in N\backslash\{i_1, i_2, \ldots, i_{k-1}\}$. We denote the cost of the path exchange P with respect to the solution ϕ by $Cost(\phi, P)$. Therefore,

$$\begin{aligned} Cost(\phi, P) &= z(\phi^P) - z(\phi) \\ &= \sum_{i\in P}\sum_{j=1}^{n} f_{ij}(d_{\phi^P(i)\phi^P(j)} - d_{\phi(i)\phi(j)}) \\ &\quad + \sum_{i\in P}\sum_{\substack{j=1\\ j\notin P}}^{n} f_{ji}(d_{\phi^P(j)\phi^P(i)} - d_{\phi(j)\phi(i)}). \qquad (7) \end{aligned}$$

Observe that ϕ^P and $\phi^{C(P)}$ differ only in the location of the facility i_k. This observation allows us to compute the cost of the cyclic exchange $C(P)$ from the cost of the path exchange P in $O(k)$ time using the following expression:

$$\begin{aligned} &Cost(\phi, C(P)) - Cost(\phi, P) \\ &\quad = c^{\phi}_{i_k i_1} + \sum_{j\in C} f_{i_k j}\big((d_{\phi(i_1)\phi^C(j)} - d_{\phi(i_k)\phi^P(j)}) \\ &\qquad\qquad - (d_{\phi(i_1)\phi(j)} - d_{\phi(i_k)\phi(j)})\big) \\ &\qquad + \sum_{\substack{j\in C\\ j\neq i_k}} f_{j i_k}\big((d_{\phi^C(j)\phi(i_1)} - d_{\phi^P(j)\phi(i_k)}) \\ &\qquad\qquad - (d_{\phi(j)\phi(i_1)} - d_{\phi(j)\phi(i_k)})\big). \qquad (8) \end{aligned}$$

Suppose that we extend the path P to $P' = i_1 - i_2 - \cdots - i_k - i_{k+1}$ by adding the node i_{k+1}. Now, we can determine the cost of the path P' from the cost of the path P in $O(k)$ time using the following expression:

$$\begin{aligned} &Cost(\phi, P') - Cost(\phi, P) \\ &\quad = c^{\phi}_{i_k i_{k+1}} + \sum_{j\in P} f_{i_k j}\big((d_{\phi(i_{k+1})\phi^{P'}(j)} - d_{\phi(i_k)\phi^P(j)}) \\ &\qquad\qquad - (d_{\phi(i_{k+1})\phi(j)} - d_{\phi(i_k)\phi(j)})\big) \\ &\qquad + \sum_{\substack{j\in P\\ j\neq i_k}} f_{j i_k}\big((d_{\phi^{P'}(j)\phi(i_{k+1})} - d_{\phi^P(j)\phi(i_k)}) \\ &\qquad\qquad - (d_{\phi(j)\phi(i_{k+1})} - d_{\phi(j)\phi(i_k)})\big). \qquad (9) \end{aligned}$$

In our enhanced version, we maintain the cost of each path P enumerated by the algorithm. Given the cost of path P, we can determine the cost of the cycle $C(P)$ in $O(k)$ time. Additionally, when we extend any path P, the cost of the extended path too can be computed in $O(k)$ time. Therefore, the running time of the generic search procedure is $O(K\sum_{k=2}^{K}|P^k|)$, plus the time required by the subroutine *PathSelect*.

4. Specific Implementations

In Section 3, we presented a generic search algorithm to identify a profitable multi-exchange. We can derive several specific implementations of the generic version by implementing the procedure *PathSelect*(P^k) differently. The procedure *PathSelect*(P^k) accepts as an input a set of paths P^k and returns a subset S^k of these paths. Next, we describe several ways in which *PathSelect* can be implemented.

4.1. Implementation 1 (All Paths)

In this version, we define *PathSelect*(P^k) as P^k itself; in other words, we select all of the paths to be taken to the next stage. This version guarantees that we will always find a profitable multi-exchange if it exists. However, the number of paths enumerated by the algorithm increases exponentially with k, and it takes too long to find profitable k-exchanges for $k \geq 6$ even for $n = 25$.

4.2. Implementation 2 (Negative Paths)

In this version, the subroutine *PathSelect*(P^k) returns only those paths that have a negative cost; in other words, *PathSelect*$(P^k) = \{P \in P^k\colon Cost(\phi, P) < 0\}$, where ϕ is the current solution. This version is motivated by the intuition that if there is a profitable multi-exchange $C = i_1 - i_2 - \cdots - i_k - i_1$, then there should exist a node in this sequence, say node i_l, so that each of the paths $i_l - i_{l+1}, i_l - i_{l+1} - i_{l+2}, \ldots, i_l - i_{l+1} - i_{l+2} - \cdots - i_{l+k}$ has a negative cost. Though results of this type are valid for many combinatorial optimization problems, it is *not* true for the QAP. However, it is a reasonable heuristic to eliminate paths that are less likely to yield profitable multi-exchanges.

4.3. Implementation 3 (Best αn^2 Paths)

In this version, we sort all the paths in P^k in nondecreasing order of path costs and select the first αn^2 paths in which α is a specified constant. For example, if $\alpha = 2$, then we select the best $2n^2$ paths. This version is motivated by the intuition that the paths with lower costs are more likely to yield profitable multi-exchanges. The choice of α allows us to strike an appropriate tradeoff between the running time and the solution quality. Higher values of α increase the chances of finding profitable multi-exchanges but also increase the time needed to find a profitable multi-exchange. Our computational results, presented in Section 7, indicate that $\alpha = 1$ is a worthwhile choice considering both the running time and solution quality. We have used the max heap data structure (Cormen et al. 2001) to keep αn^2 paths in a stage. Therefore, if there are pn possible paths (as discussed in Section 3), then it takes $pn\log(pn)$ time to store αn^2 best paths in a heap.

4.4. Implementation 4 (Best n Paths)

In this implementation, we select the best path in P^k beginning with node i for each $1 \le i \le n$. Therefore, the set S^k contains at most one path starting at each node in N. Note that in Implementation 3, it is possible that many low-cost paths contain the same set of arcs, making the search less diverse. Allowing each node to be the starting point of a different path can add some diversity to the heuristic search process.

We applied Implementations 3 and 4 of *PathSelect* to the test problems. Our experiment with the test problems indicates that it is difficult to use Implementation 1 even for moderately-sized problems. Moreover, in all cases, Implementation 3 provides better results than Implementation 2.

5. The Neighborhood Search Algorithm

In this section, we describe our neighborhood search algorithm for the QAP (see Figure 3). Our algorithm starts with a random permutation, which we obtain by generating pseudorandom numbers between 1 and n and rejecting the numbers already generated. The algorithm successively improves the current solution by performing profitable multi-exchanges, which are obtained with the K-exchange search procedure, until the procedure fails to produce a profitable multi-exchange.

We will now perform the running-time analysis of the algorithm. The initial construction of the improvement graph takes $O(n^3)$ time. The time required by the K-exchange search is $O(K^2p)$, where p is the maximum number of paths maintained by the procedure during any iteration (see Section 3). For Implementation 3 of *PathSelect*, $p \le \alpha n^3$, and this procedure requires $O(K^2n^3)$ time per iteration (that is,

```
algorithm QAP-neighborhood-search;
begin
  generate an initial random permutation $\phi$;
  construct the improvement graph $G(\phi)$;
  while K-exchange search returns a nonempty
    multi-exchange $C$ do
  begin
    replace the permutation $\phi$ by the permutation $\phi^C$;
    update the improvement graph;
  end;
  return the permutation $\phi$;
end;
```

Figure 3 The Neighborhood Search Algorithm for the QAP

per improvement). For Implementation 4 of the *PathSelect*, $p \le n$, and the procedure again takes a time of $O(K^2n^2)$. Updating the improvement graph takes $O(n^2K)$ time (see Section 2).

Each execution of the *QAP-neighborhood-search* algorithm yields a locally optimal solution of the QAP with respect to the neighborhood defined by the *K-exchange search* procedure. The solution obtained depends upon the initial random permutation ϕ and the version of the *PathSelect* procedure that we use. We refer to one execution of the algorithm as one *run*. Our computational investigations revealed that if we apply only one run of the algorithm, then the solution methods are not very robust. The QAP in general has an extremely large number of locally optimal solutions. Each run produces a locally optimal solution that may be viewed as a random sample in the set of locally optimal solutions. To obtain a robust, locally optimal solution, we need to perform several runs of the algorithm and employ the best locally optimal solution found in these runs.

6. Accelerating the Search Algorithm

In this section, we describe a method to speed up the performance of the generic search algorithm and its specific implementations. To perform this acceleration, we acknowledge that several paths provide the same multi-exchange. For example, all the paths $i_1 - i_2 - i_3 - i_4$, $i_2 - i_3 - i_4 - i_1$, $i_3 - i_4 - i_1 - i_2$, and $i_4 - i_1 - i_2 - i_3$ imply the same multi-exchange $i_1 - i_2 - i_3 - i_4 - i_1$ when we connect the last node of these paths to the first node of the path. In general, a k-exchange can be represented by k different paths. Since our generic search algorithm enumerates k-exchanges by enumerating paths, we may obtain the same k-exchange several times during the search process through different paths. To avoid repeated enumeration of the multi-exchanges, our search algorithm maintains certain kinds of paths, called *valid paths*, which are defined as follows:

Valid Paths: *A path $i_1 - i_2 - \cdots - i_k$ is a valid path if $i_1 \le i_j$ for every $2 \le j \le k$.*

Our generic search algorithm enumerates only valid paths. The following lemma reveals that we do not miss any multi-exchanges by maintaining valid paths only. The proof of the lemma is obvious.

Lemma 2. *Any multi-exchange can be enumerated by maintaining only valid paths.*

We can easily modify the generic search algorithm so that it only enumerates valid paths. In this modified algorithm, when we consider adding the arc (i_k, i_{k+1}) to the path $i_1 - i_2 - \cdots - i_k$, we compare i_1 with i_{k+1}. If $i_1 \le i_{k+1}$, then we add the arc; otherwise, we do not add it. The above lemma holds if we enumerate all paths. However, as we keep only αn^2 paths in each stage, there may be cases when we might miss a profitable multi-exchange. Our experiment demonstrates that the loss in missed improvements is well compensated by the gain in time. The computational results presented in Section 7 reveal that enumerating only valid paths substantially decreases the running time of the generic search algorithm.

7. Computational Testing

In this section, we describe the computational results of our neighborhood search algorithms. We implemented all of our algorithms in C and tested them on a Pentium IV machine (with a processor speed of 2.4 GHz). We tested the algorithms on 142 benchmark instances available from QAPLIB (2005). Our computational results include analyzing the CPU times and the quality of the solutions as well as understanding the behavior of the VLSN search algorithms.

Neighborhood search algorithms require a feasible solution as the starting solution. So, we generated random permutations of n numbers and used them as starting solutions. We implemented a multi-start version of the neighborhood search algorithm in which we applied the neighborhood search algorithm multiple times with different starting solutions, called *runs*. Next, we selected the best solution found in these runs. The number of runs depends on the size of the problem instance.

In Section 4, we propose four implementations of the generic VLSN search algorithm for the QAP. The first implementation maintains all the paths enumerated in the search process. We found that the number of paths grows very quickly with k and that the algorithm runs very slowly even when we go up to k-exchanges with $k = 6$. For example, to solve a QAP with $n = 42$ (problem sko42), each run of this implementation takes about 8 seconds for $k = 4$. However, Implementation 3 takes only 0.025 second per run. Additional preliminary tests revealed that this implementation is not as competitive as other implementations so we decided not to perform a thorough testing of the first implementation.

In the second implementation of the VLSN search algorithm, we maintain only those paths that have negative costs. For many combinatorial optimization problems, maintaining only negative-cost paths is sufficient to enumerate negative-cost cycles (improved neighbors), but this is not true for the QAP due to the cost structure's nonlinearity. Our initial computational testing revealed that maintaining only negative-cost paths is not an efficient heuristic to enumerate negative-cost cycles. Therefore, we did not perform a thorough testing of this implementation.

Our preliminary testing also revealed that Implementations 3 and 4 exhibited the best overall behavior and each deserved a thorough testing. The following details of Implementation 3 are worth mentioning. Recall from Section 4 that we retain only the αn^2 best paths in P^k. We used the Max Heap data structure (Cormen et al. 2001) to store these paths. In so doing, we found that $\alpha = 1$ provides fairly good results, and so we used this value of α. In addition, we used only those paths whose cost is not more than 0.5% of the best objective function value of the QAP found so far. We observed that different values of the cost multiplier are suitable for different instances; however, 0.5% seemed to be the best across the largest number of test instances. We determined that using higher-cost paths rarely leads to negative-cost cycles. Finally, when we examined paths in P^k to enumerate cycles of length k and found several negative-cost cycles, we used the least-cost negative cycle to obtain the next solution. We applied Implementation 4 in a straightforward fashion; however, before enumerating paths, we eliminated all the negative cycles of length 2 by performing 2-exchanges.

7.1. Analysis of the Solution Quality

We applied Implementations 3 and 4 to the 142 benchmark instances in QAPLIB, of which 108 were symmetric. We applied multiple runs of each implementation and ran them for a specified amount of time. For the symmetric instances, we ran our algorithm for one hour for $n \le 40$ and for two hours for $n > 40$. The running times for the asymmetric instances were 1.5 hours for $n \le 40$ and three hours for $n > 40$. In our test, we chose the parameter values $k = 4$ and $\alpha = 1$. The reasons for choosing these values are explained later in this section. Tables A1 and A2 in the Online Supplement to this paper on the journal's website display the results of these algorithms for symmetric and asymmetric instances and compare our solutions with the solutions obtained by the 2-exchange algorithm (2OPT) and the best known solutions (BKS). The columns titled "%Gap" and "nRuns," respectively, provide the percent deviation of the best solution found in all the runs with respect to the best known solution and the number of runs. We conclude:

- Implementation 3 exhibited the best overall performance. It obtained the best known solutions in 77 out of 108 symmetric instances and in 24 out of 34 asymmetric instances. Its average error was the lowest, and it found the best known solutions with the maximum frequency.
- Implementation 3 exhibited superior performance when compared to 2OPT in terms of the gap between the best solution found by the algorithm with the best known solution. For 30 symmetric instances, Implementation 3 obtained better solutions than 2OPT, and for three symmetric instances 2OPT obtained better solutions. Similarly, for ten asymmetric instances, Implementation 3 obtained better solutions than 2OPT, and for one asymmetric instance 2OPT obtained better solutions.
- Implementation 4 also exhibited superior performance with respect to 2OPT, but its overall performance was worse than Implementation 3.

The above results seem to suggest that the VLSN search heuristic is more effective overall than the traditional 2-exchange neighborhood search heuristic. When both algorithms are run for the same amount of time, 2OPT performs many more runs yet its best solution is, on average, worse than that of the VLSN search. Therefore, the extra time per iteration required by the VLSN search algorithm is more than justified by the better quality of the solutions obtained. We will now describe some computational investigations that we performed in order to understand the behavior of our implementations.

7.2. Effect of Neighborhood Size

In our approach, the size of the neighborhood critically depends upon (i) the maximum cycle length and (ii) the number of maintained paths of a given length. The larger the cycle length and the number of paths maintained, the larger the neighborhood, the greater the running time, and the better the quality of the solution obtained (in general). Therefore, it is worthwhile to examine the effect of these two parameters on the running time and the solution quality.

In our first experiment, we considered six problems of the same size; sko100a, sko100b, sko100c, sko100d, sko100e, and sko100f, and we applied 100 runs of Implementation 3 with cycle lengths varying from two to seven. Additionally, we noted the average running time required by the algorithm (per run) and the average gap (per run). We fixed the number of paths maintained by the algorithm at n^2. Figure 4 plots these two values as a function of cycle length. Clearly, the average gap decreases significantly with the increase in cycle length until it reaches four. After that, the average gap does not change much. We also observe that the running time of the algorithm increases linearly with the increase in the cycle length. For this

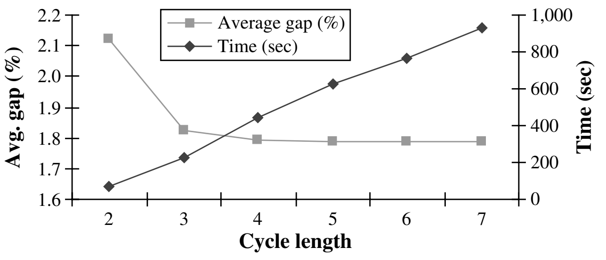

Figure 4 The Effect of Cycle Length for 100 Runs of Problems sko100a–f

reason, we think that the cycle length of four strikes an appropriate balance between the solution accuracy and the solution time. We therefore used this value in the computational results.

Our second experiment was similar to the first except that we varied the number of paths maintained by the algorithm while keeping the cycle length fixed at four. Figure 5 plots the average gap and the average time per run when we performed 100 runs of Implementation 3 on problems sko100a–f. We observe that the solution accuracy gradually improves as the number of paths increases. Also, the running time of the algorithm increases linearly with the number of paths maintained. We believe that maintaining n^2 paths is an effective compromise between solution quality and solution time, and so we used this value in our experiments.

In another experiment, we counted the number of improving iterations with cycle lengths 2, 3, and 4. Recall that our algorithm performs a 3-exchange when it fails to find a 2-exchange, and it performs a 4-exchange when it fails to find a 3-exchange. Table 1 provides the number of iterations of 2-, 3-, and 4-exchanges for ten benchmark instances in which we apply 100 runs of Implementation 3. We selected these instances as representative instances as they belong to different instance classes and are of different sizes. We therefore observe that there are many more iterations with 2-exchanges compared to 3-exchanges and many more 3-exchanges compared to 4-exchanges.

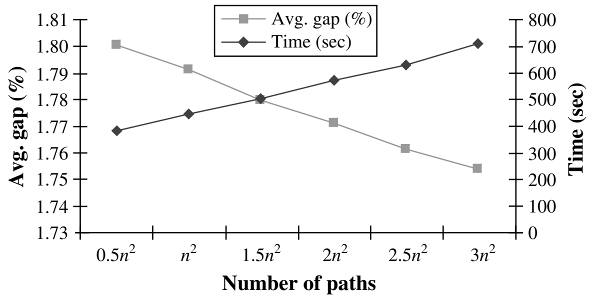

Figure 5 The Effect of the Number of Paths for 100 Runs of Problems sko100a–f

Table 1 **Number of Iterations of Cycle Lengths Two, Three, and Four**

	Number of iterations		
Problem	2	3	4
chr22a	1,614	151	49
kra30a	2,283	123	30
kra30b	2,306	125	32
nug30	2,580	104	42
ste36a	3,537	142	44
tho40	3,839	124	35
wil50	5,298	81	38
sko42	4,246	150	64
sko100a	13,232	270	70
tai100a	7,267	274	75

Table 2 **The Effects of the Accelerated Path Enumeration Scheme**

	Using speedup technique		Without speedup technique	
Problem	Average gap	Time in seconds	Average gap	Time in seconds
chr22a	10.00	1	9.13	5
kra30a	6.44	5	6.27	12
kra30b	4.26	5	4.05	12
nug30	3.19	5	2.92	12
ste36a	9.34	10	8.37	27
tho40	3.87	12	3.76	28
wil50	1.54	24	1.41	70
sko42	2.68	17	2.57	40
sko100a	1.87	283	1.78	962
tai100a	2.88	280	2.48	1,191

7.3. Effect of the Speedup Technique

The reader may recall from Section 6 that we employed a speedup technique to reduce redundant enumeration of cycles. In this technique, we maintain only those valid paths $i_1, i_2, \ldots, i_k$ for which $i_k > i_1$. Lemma 2 revealed that we will not miss any negative cycles even if we maintain only valid paths. This proof relied on the assumption that we maintain *all* valid paths. Since our algorithm maintains only n^2 paths, we might miss some negative cycles, and the speedup technique may deteriorate the quality of the solutions obtained. We performed an experiment to assess the effect of the speedup technique on the solution quality and solution time. Table 2 provides average gap and computational time for ten benchmark instances. We applied 100 runs in each benchmark instance and noted the average values. We observe that the speedup technique decreases the running time substantially but also worsens the solution quality. Therefore, we believe that overall it is advantageous to incorporate the speedup technique since the saved time can be used to perform more runs of the algorithm and improve its overall performance.

8. Conclusions

We developed a very large-scale neighborhood structure for the QAP. We demonstrated that using the concept of the improvement graph, we can easily and quickly enumerate multi-exchange neighbors of a given solution. We developed a generic search procedure to enumerate and evaluate neighbors and proposed several specific implementations of the generic procedure. We additionally performed extensive computational investigations of our implementations; these suggest that multi-exchange neighborhoods are superior to the commonly used 2-exchange neighborhoods.

Our implementations of multi-exchange neighborhood search algorithms are local improvement methods. We wanted the focus of our research effort to be more on neighborhood structure and less on specific implementations. There are possibilities for further improvement using ideas from tabu search (Glover and Laguna 1997), which we leave as a topic of future research. Additionally, our neighborhood structure may be useful within a genetic algorithm to improve the quality of a population's individuals (Ahuja et al. 2000, Drezner 2003).

Acknowledgments

The authors thank the referees and the associate editor for their insightful comments and suggestions, which greatly improved the readability of this paper. This project was partly funded by National Science Foundation (NSF) Grants DMI-9900087, DMI-9820998, DMI-0217359, and DMI-0217123, and Office of Naval Research (ONR) Grant N00014-98-1-0317.

References

Ahuja, R. K., T. L. Magnanti, J. B. Orlin. 1993. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Eaglewood Cliffs, NJ.

Ahuja, R. K., J. B. Orlin, D. Sharma. 2001a. Multi-exchange neighborhood search algorithms for the capacitated minimum spanning tree problem. *Math. Programming* **91** 71–97.

Ahuja, R. K., J. B. Orlin, D. Sharma. 2001b. A composite very large-scale neighborhood structure for the capacitated minimum spanning tree problem. *Oper. Res. Lett.* **31** 185–194.

Ahuja, R. K., J. B. Orlin, A. Tiwari. 2000. A greedy genetic algorithm for the quadratic assignment problem. *Comput. Oper. Res.* **27** 917–934.

Ahuja, R. K., Ö. Ergun, J. B. Orlin, A. P. Punnen. 2002. A survey of very large scale neighborhood search techniques. *Discrete Appl. Math.* **123** 75–102.

Ahuja, R. K., J. Goodstein, A. Mukherjee, J. B. Orlin, D. Sharma. 2007. A very large-scale neighborhood search algorithm for the combined through-fleet-assignment model. *INFORMS J. Comput.* **19** 416–428.

Anstreicher, K. M., N. W. Brixius, J. P. Goux, J. Linderoth. 2002. Solving large quadratic assignment problems on computational grids. *Math. Programming* **91** 563–588.

Bazaraa, M. S., M. D. Sherali. 1980. Benders' partitioning scheme applied to a new formulation of the quadratic assignment problem. *Naval Res. Logist. Quart.* **27** 29–41.

Buffa, E. S., G. C. Armour, T. E. Vollmann. 1964. Allocating facilities with CRAFT. *Harvard Bus. Rev.* **42** 136–158.

Burkard, R. E., T. Bönniger. 1983. A heuristic for quadratic boolean programs with applications to quadratic assignment problems. *Eur. J. Oper. Res.* **13** 374–386.

Burkard, R. E., E. Çela, P. M. Pardalos, L. S. Pitsoulis. 1998. The quadratic assignment problem. D. Z. Du, P. M. Pardalos, eds. *Handbook of Combinatorial Optimization*, Vol. 3. Kluwer Academic Publishers, Boston, MA, 241–337.

Çela, E. 1998. *The Quadratic Assignment Problem—Theory and Algorithms*. Kluwer Academic Publishers, Boston, MA.

Christofides, N., E. Benavent. 1989. An exact algorithm for the quadratic assignment problem. *Oper. Res.* **37** 760–768.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, C. Stein. 2001. *Introduction to Algorithms*, 2nd ed. MIT Press, Cambridge, MA.

Deineko, V. G., G. J. Woeginger. 2000. A study of exponential neighborhoods for the travelling salesman problem and for the quadratic assignment problem. *Math. Programming* **87** 519–542.

Drezner, Z. 2003. A new genetic algorithm for the quadratic assignment problem. *INFORMS J. Comput.* **15** 320–330.

Ergun, Ö. 2001. New neighborhood search algorithms based on exponentially large neighborhoods. Doctoral thesis, Operations Research Center, MIT, Cambridge, MA.

Fleurent, C., J. A. Ferland. 1994. Genetic hybrids for the quadratic assignment problem. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 16. American Mathematical Society, Providence, RI. 173–187.

Gendreau, M., F. Guertin, J.-Y. Potvin, R. Seguin. 1998. Neighborhood search heuristics for a dynamic vehicle dispatching problem with pick-ups and deliveries. CRT Research Report CRT-98-10, Centre for Research on Transportation, University of Montreal, Montreal, Canada.

Glover, F. 1996. Ejection chains, reference structures and alternating path methods for traveling salesman problems. *Discrete Appl. Math.* **65** 223–253.

Glover, F., M. Laguna. 1997. *Tabu Search*. Kluwer Academic Publishers, Boston, MA.

Hahn, P. M., W. L. Hightower, T. A. Johnson, M. Guignard-Spielberg, C. Roucairol. 2001. Tree elaboration strategies in branch and bound algorithms for solving the quadratic assignment problem. *Yugoslavian J. Oper. Res.* **11** 41–60.

Kelly, J. P., J. Xu. 1999. A set-partitioning-based heuristic for the vehicle routing problem. *INFORMS J. Comput.* **11** 161–172.

Lawler, E. L. 1963. The quadratic assignment problem. *Management Sci.* **9** 586–599.

Li, Y., P. M. Pardalos, M. G. C. Resende. 1994. A greedy randomized adaptive search procedure for the quadratic assignment problem. P. M. Pardalos, H. Wolkowicz, eds. *Quadratic Assignment and Related Problems. DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Providence, RI, 237–261.

Malucelli, F. 1993. *Quadratic Assignment Problems: Solution Methods and Applications*. Dipartimento di Informatica, Università di Pisa, Italy.

Maniezzo, V., A. Colorni, M. Dorigo. 1994. The ant system applied to the quadratic assignment problem. Technical Report IRIDIA/94-28, Université libre de Bruxelles, Brussels, Belgium.

Müller-Merbach, H. 1970. *Optimale Reihenfolgen*. Springer Verlag, Berlin, Germany, 158–171.

Pardalos, P. M., J. Crouse. 1989. A parallel algorithm for the quadratic assignment problem. *Proc. Supercomputing 1989 Conf.*, ACM Press, New York, 351–360.

Pardalos, P. M., F. Rendl, H. Wolkowicz. 1994. The quadratic assignment problem. P. M. Pardalos, H. Wolkowicz, eds. *Quadratic Assignment and Related Problems. DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Providence, RI, 1–42.

QAPLIB. 2005. QAPLIB home page, http://www.seas.upenn.edu/qaplib/.

Rego, C., C. Roucairol. 1996. Parallel tabu search algorithm using ejection chains for the vehicle routing problem. I. H. Osman, J. P. Kelly, eds. *Metaheuristics: Theory and Applications*. Kluwer Academic Publishers, Boston, MA, 661–675.

Skorin-Kapov, J. 1990. Tabu search applied to the quadratic assignment problem. *ORSA J. Comput.* **2** 33–45.

Taillard, E. 1991. Robust tabu search for the quadratic assignment problem. *Parallel Comput.* **17** 443–455.

Talluri, K. T. 1996. Swapping applications in a daily fleet assignment. *Transportation Sci.* **31** 237–248.

Tate, D. E., A. E. Smith. 1995. A genetic approach to the quadratic assignment problem. *Comput. Oper. Res.* **22** 73–83.

Thompson, P. M., J. B. Orlin. 1989. The theory of cyclic transfers. Operations Research Center Working Paper, MIT, Cambridge, MA.

Thompson, P. M., H. N. Psaraftis. 1993. Cyclic transfer algorithms for multi-vehicle routing and scheduling problems. *Oper. Res.* **41** 935–946.

West, D. H. 1983. Algorithm 608: Approximate solution of the quadratic assignment problem. *ACM Trans. Math. Software* **9** 461–466.

Wilhelm, M. R., T. L. Ward. 1987. Solving quadratic assignment problems by simulated annealing. *IEEE Trans.* **19** 107–119.