MIT Sloan School of Management

Working Paper 4386-02
July 2002

VERY LARGE-SCALE NEIGHBORHOOD SEARCH FOR THE QUADRATIC ASSIGNMENT PROBLEM

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

Very Large-Scale Neighborhood Search for the Quadratic Assignment Problem

Ravindra K. Ahuja[1], Krishna C. Jha[2], James B. Orlin[3], and Dushyant Sharma[4]

Abstract

The Quadratic Assignment Problem (QAP) consists of assigning n facilities to n locations so as 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 size instances (typically, $n \leq 25$). Neighborhood search algorithms are the most popular heuristic algorithms to solve larger size instances of the QAP. The most extensively used neighborhood structure for the QAP is the 2-exchange neighborhood. This neighborhood is obtained by swapping the locations of two facilities and thus has size O(n^2). Previous efforts to explore larger size 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 where the size of the neighborhood is very large and we propose a novel search procedure to heuristically enumerate good neighbors. Our search procedure relies on the concept of improvement graph which allows us to evaluate neighbors much faster than the existing methods. We present extensive computational results of our algorithms on standard benchmark instances. These investigations reveal that very large-scale neighborhood search algorithms give consistently better solutions compared the popular 2-exchange neighborhood algorithms considering both the solution time and solution accuracy.

---

[1] Ravindra K. Ahuja, Industrial and Systems Engg., University of Florida, Gainesville, FL 32611, USA.
[2] Krishna C. Jha, Industrial and Systems Engg., University of Florida, Gainesville, FL 32611, USA.
[3] James B. Orlin, Sloan School of Management, MIT, Cambridge, MA 02139, USA.
[4] Dushyant Sharma, Operations Research Center, MIT, Cambridge, MA 02139, USA.

1. INTRODUCTION

The Quadratic Assignment Problem (QAP) is a classical combinatorial optimization problem and is widely regarded as one of the most difficult problems in this class. Given a set N = {1, 2, ..., n}, and nxn matrices $F = \{f_{ij}\}$, $D = \{d_{ij}\}$, and $B = \{b_{ij}\}$, the QAP is to find a permutation ϕ of the set N which minimizes:

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

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 campus, 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 feasible solutions of QAP and permutations ϕ.

Observe that (1) consists of two terms. The first term 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). The second term considers the cost of erecting facilities which may be location-dependent. The matrix $B = \{b_{ij}\}$ represents the cost of creating facility i at location j. Hence, the QAP is to find an assignment of facilities to locations so as to minimize the total cost of flow between the facilities and the cost of erecting the facilities. The matrices F and D are typically symmetric matrices but are not required to be so. In our algorithms, we allow asymmetric instances and thus do not assume that $f_{ij} = f_{ji}$ or $d_{ij} = d_{ji}$. However, for the sake of simplicity, we will assume in Sections 2 through 6 that we are working with symmetric QAPs. In Section 7, we will study asymmetric QAPs.

In addition to the facility layout, the QAP arises in many other applications, such as the allocation of plants to candidate locations, backboard wiring problem, design of control panels and typewriter keyboards, turbine balancing, ordering of interrelated data on a magnetic tape, and others. The details and references for these and additional applications can be found in Malucelli [1993], Pardalos, Rendl and Wolkowicz [1994], Burkard et al. [1998], and Cela [1998]. Given the wide range of applications and the difficulty of solving the problem, the QAP has been investigated extensively by the research community. 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 (Lawler [1963], Pardalos and Crouse [1989]). Among these, the branch and bound algorithms are the most successful, but they are generally unable to solve problems of size larger than $n = 25$.

Since the applications of the QAP often give rise to problems of size far greater than 25, there is a need for good heuristics for QAP that can solve larger size 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, Armour and Vollmann [1964], Muller-Merbach [1970]); (ii) *limited enumeration methods* (West [1983], Burkard and Bonniger [1983]); (iii) *GRASP* (greedy randomized adaptive search procedure) (Li, Pardalos, and Resende [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 [1985], Ahuja, Orlin, and Tewari [1998], Drezner [2001]); and (vii) *ant systems* (Maniezzo, Colorni, and Dorigo [1994]). The tabu search method of Taillard [1991], the GRASP method of Li, Pardalos, and Resende [1994], and the genetic algorithm by Drezner [2001] are the most accurate heuristics among these methods.

As observed in the survey paper of Burkard et al. [1998], the current neighborhood search meta-heuristic (tabu search and simulated annealing) algorithms for the QAP use the *2-exchange* neighborhood structure in the search. 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 in the permutation ϕ. It is easy to see that the number of 2-exchange neighbors of a permutation is $O(n^2)$. There has been very limited effort in the past to explore larger neighborhood structures for the QAP as the time needed to identify an improved neighbor becomes too high. In this paper, we investigate the neighborhood structure based on *multi-exchanges*, which is a natural generalization of the 2-exchanges. A multi-exchange is specified by a cyclic sequence $C = i_1$ - i_2 - … - 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 $\phi(i_3)$, and so on, and finally facility i_k is assigned to $\phi(i_1)$. The location of all other facilities is not changed. We denote by ϕ^C the permutation obtained by applying the multi-exchange C to the permutation ϕ. In other words,

$$\phi^C(i) = \phi(i) \text{ for } i \in N \setminus \{i_1, \ldots, i_k\},$$
$$\phi^C(i_p) = \phi(i_{p+1}) \text{ for } p = 1, \ldots, k\text{-}1, \text{ and} \tag{2}$$
$$\phi^C(i_k) = \phi(i_1).$$

We define the length of a multi-exchange as the number of facilities involved in the corresponding cyclic sequence. For example, the cyclic sequence $C = i_1$ - i_2 - … - i_k - i_1 has length k. We also refer to a multi-exchange of length k as 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*.
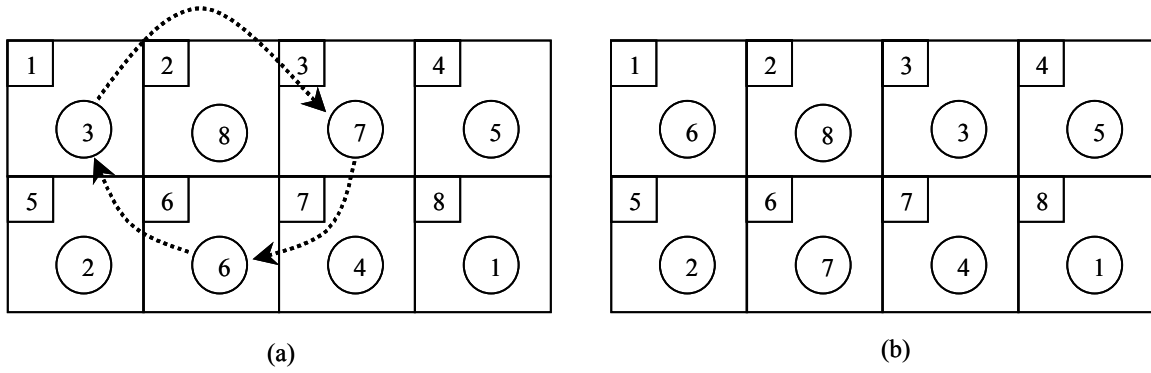
Figure 1. (a) Initial assignment of facilities to locations.
(b) Assignment after the cyclic exchange 3-7-6-3.

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 at most K. We note that the two-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 number 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 as 6×10^{18} neighbors. This neighborhood structure falls under the category of very large-scale neighborhood (VLSN) structures where 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 in this area). One of the tools used in performing search over very large-scale neighborhood structures is the concept of the *improvement graph*. In this technique, we associate a graph, called the improvement graph $G(\phi)$, 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 to some 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 problem). The concept of the improvement graph was first proposed by Thompson and Orlin [1989] for a partitioning problem, where a set of elements is partitioned into several subsets of elements so as 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 vehicle routing problem (Thompson and Psaraftis [1993], Ibaraki et al. [2002]) and the capacitated minimum spanning tree problem (Ahuja, Orlin, Sharma [2001a, 2001b]). The concept of improvement graph was also

used by Talluri [1996] and Ahuja et al. [2001c] to search very large-scale neighborhoods as in fleet assignment problems arising in airline scheduling. Ergun [2001] also proposed several improvement graphs for the vehicle routing problem and machine scheduling problems.

In this paper, we study the use of the improvement graph for the multi-exchange neighborhood structure for the QAP. However, our current application of the improvement graph is different than previous applications. 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 allows us to enumerate good neighbors quickly. The improvement graph also allows us to evaluate the cost of a neighbor faster than using a normal method. Typically, evaluating a k-exchange neighbor for the QAP takes O(nk) time; but using the improvement graph we can do it in O(k) average time per neighbor.

We developed a generic search procedure to enumerate neighbors using improvement graphs. We also developed several implementations of the generic search procedure which enumerate the neighborhoods exactly as well as heuristically. We present a detailed computational investigation of local improvement algorithms based on our neighborhood search structures. Our investigations yield the following conclusions: (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 entire neighborhood and can develop improvements that are almost as good.

This paper is organized as follows. In Section 2, we describe the improvement graph data structure for the QAP. We present a generic heuristic search procedure for the K-exchange neighborhood structure for the QAP in Section 3. In Section 4, we describe 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 we use to speed up the performance of the algorithm. For the simplicity of derivations, in Sections 2 through 6, we analyze and discuss the algorithms for symmetric cases only, which can be easily generalized for asymmetric cases. In Section 7, we present the corresponding expressions for the general case (both symmetric and asymmetric instances). We provide and analyze the computational results from our implementations in Section 8. Section 9 summarizes our contributions.

2. IMPROVEMENT GRAPH

One of the main contributions of this paper is the 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. This section

5

as well as the following sections requires some network notations, such as cycles and paths. We will use the graph notation given in the book by Ahuja, Magnanti, and Orlin [1993] and refer the reader to this book for the same.

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, that is,

$$Cost(\phi, C) = z(\phi^C) - z(\phi) = 2 * \sum_{i \in C} \sum_{j=1}^{n} f_{ij} \left(d_{\phi^C(i)\phi^C(j)} - d_{\phi(i)\phi(j)} \right). \tag{3}$$

Clearly, the cost of the k-exchange C can be computed in time $O(kn)$. We will show that 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 multi-exchanges substantially faster than standard methods. In fact, it also leads to dramatic improvements in the running time to identify traditional 2-exchanges.

We associate an improvement graph $G(\phi) = (N, A)$ with ϕ, which is a directed graph comprising of the node set N and the 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 $C = i_1 - i_2 - \dots - i_k - i_1$ defines a (directed) cycle $i_1 - i_2 - \dots - i_k - i_1$ in $G(\phi)$ and, similarly, each (directed) cycle $i_1 - i_2 - \dots - i_k - i_1$ in $G(\phi)$ defines a multi-exchange $i_1 - i_2 - \dots - i_k - i_1$ with respect to ϕ. Thus, there is a one-to-one correspondence between multi-exchanges with respect to ϕ and cycles in $G(\phi)$. We will, henceforth, use C to denote both a multi-exchange and a cycle in $G(\phi)$, and its type will be apparent from the 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 $C = i_1 - i_2 - \dots - i_k - i_1$, 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 = NP because the multi-exchange neighborhood structure includes all feasible solutions for an instance of the QAP. 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: it is 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 other facilities do not move. Observe that this change indicates that after the change there is no facility at location $\phi(i)$

6

and the location $\phi(j)$ has two facilities. Thus, to determine the cost of the change, we need to take the difference between the costs of interactions between facility i and other facilities, before and after the change. Let ϕ' denote the solution after the change. Then, $\phi'(l) = \phi(l)$ for $l \ne i$ and $\phi'(i) = \phi(j)$. Note that ϕ' is not a permutation because $\phi'(i) = \phi(j)$. We define $c_{ij}^\phi = z(\phi') - z(\phi)$. Thus,

$$c_{ij}^\phi = z(\phi') - z(\phi) = 2 * \sum_{l=1}^{n} f_{il}\left(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)}\right), \tag{4}$$

which captures the change in the cost of interaction *from* facility i to 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 all other facilities do not move. But in the multi-exchange C several facilities move and we do not correctly account for the cost of flow between facilities in C. We, however, correctly account for the cost of flow between any two facilities if one of the two facilities is not in C. We show next that the cost term $Cost(\phi, C)$ can be computed by adding a corrective term to $\sum_{(i,j)\in C} c_{ij}^\phi$.

$$
\begin{aligned}
Cost(\phi, C) = z(\phi^C) - z(\phi) &= 2 * \sum_{i\in C}\sum_{j=1}^{n} f_{ij}\left(d_{\phi^C(i)\phi^C(j)} - d_{\phi(i)\phi(j)}\right) \\
&= 2 * \sum_{i\in C}\sum_{j\notin C} f_{ij}\left(d_{\phi^C(i)\phi(j)} - d_{\phi(i)\phi(j)}\right) + 2 * \sum_{i\in C}\sum_{j\in C} f_{ij}\left(d_{\phi^C(i)\phi^C(j)} - d_{\phi(i)\phi(j)}\right) \\
&= 2 * \sum_{(i,j)\in C} c_{ij}^\phi - 2 * \sum_{i\in C}\sum_{j\in C} f_{ij}\left(d_{\phi^C(i)\phi(j)} - d_{\phi(i)\phi(j)}\right) \\
&\quad + 2 * \sum_{i\in C}\sum_{j\in C} f_{ij}\left(d_{\phi^C(i)\phi^C(j)} - d_{\phi(i)\phi(j)}\right). \tag{5}
\end{aligned}
$$

The equation (5) shows 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 term given in equation (5). This corrective term can be computed in time $O(k^2)$ if C is a k-exchange.

Let us now remark on the usefulness of the improvement graph. 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 the average) will be substantially smaller than the total cost and the

cost of the cycle C in $G(\phi)$ will be a good 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 about 100 of them are counted incorrectly which is only 1% of the total interactions between facilities. Thus, the improvement graph allows us to enumerate extremely large set of neighbors quickly using approximate costs, and the approximation in costs is quite small.

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 takes $O(kn)$ time as we would need to update the cost interactions between k facilities (that move) with other facilities. However, using (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 substantial difference in an algorithm's performance.

The benefits we derive from the use of 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 show in the next lemma that updating the improvement graph following a k-exchange takes only $O(kn^2)$ time. We also show in Section 8 that our neighborhood search algorithms use small values of k $(4$ and $5)$ only as on the benchmark instances higher values do not add extra benefit. Hence, it takes $O(n^2)$ time to update the improvement graph, which is quite efficient in practice. Thus, the time needed to construct and update the improvement graph is relatively small, and is well justified by the savings 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 $j \in C$, and (ii) $i \notin C$ and $j \notin C$. There are $2k(n - k) = O(nk)$ arcs of type (i), and $O(n^2)$ arcs of type (ii). Using (4), we can determine the cost of a type (i) arc in $O(n)$ time, thus giving a total time of (n^2k) to compute the cost of all type (i) arcs. We show next 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.

$$
\begin{aligned}
c_{ij}^{\phi^C} &= 2 * \sum_{l=1}^{n} f_{il}\left(d_{\phi(j)\phi^C(l)} - d_{\phi(i)\phi^C(l)}\right) \\
&= 2 * \sum_{l \notin C} f_{il}\left(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)}\right) + 2 * \sum_{l \in C} f_{il}\left(d_{\phi(j)\phi^C(l)} - d_{\phi(i)\phi^C(l)}\right)
\end{aligned}
$$

8

$$= 2 * \sum_{l=1}^{n} f_{il} \left(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)} \right) - 2 * \sum_{l \in C} f_{il} \left(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)} \right)$$

$$+ 2 * \sum_{l \in C} f_{il} \left(d_{\phi(j)\phi^C(l)} - d_{\phi(i)\phi^C(l)} \right)$$

$$= 2 * c_{ij}^{\phi} - 2 * \sum_{l \in C} f_{il} \left(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)} \right) + 2 * \sum_{l \in C} f_{il} \left(d_{\phi(j)\phi^C(l)} - d_{\phi(i)\phi^C(l)} \right). \tag{6}$$

Since we already know c_{ij}^{ϕ} and C, and C is a k-exchange, 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 keep the number of multi-exchanges enumerated manageable, 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 we wish to perform. This enumeration scheme is motivated by the consideration that we look for larger size multi-exchanges when smaller size 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 $P = i_1 - i_2 - \ldots - i_k$ in the improvement graph has a corresponding cycle in the improvement graph $i_1 - i_2 - \ldots - i_k - i_1$ 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 some paths of length k in $G(\phi)$. We start with $S^l = \{1, 2, \ldots, n\}$, which is the set of n paths of length 1, each consisting of a singleton node.

***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 $P = i_1 - i_2 - \ldots - i_{k-1}$, we add the arc (i_{k-1}, i_k) for each $i_k \in N \backslash \{i_1, i_2, \ldots, i_{l-1}\}$, and obtain several paths of length k. Let $E(P)$ denote the set of all paths obtained by extending the path P. Further, let $\mathcal{P}^k = \bigcup_{P \in S^{k-1}} E(P)$.

Cycle Evaluation: Each path $P \in \mathcal{P}^k$ 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 yes, we return the best multi-exchange and stop; otherwise we proceed further.

Path Pruning: In this step, we prune several paths in the set $\mathcal{P}^k$ which are less likely to lead to profitable multi-exchanges. We call a procedure, *PathSelect*($\mathcal{P}^k$), that takes as an input the set of paths $\mathcal{P}^k$ enumerated in the previous step and selects a subset S^k of it. This subset of paths will be 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 to keep the number of paths enumerated manageable.

The following algorithmic description summarizes the steps of our heuristic search procedure, which we call the *K-exchange search* procedure.

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 non-empty **and** $k < K$ **and** $W^* \geq 0$ **do**
    begin
        $k \leftarrow k+1$;
        $\mathcal{P}^k \leftarrow \bigcup_{P \in S^{k-1}} E(P)$;
        let $P_{min} \in S^k$ be the path such that $Cost(\phi, C(P_{min})) = \min\{Cost(\phi, C(P)): P \in \mathcal{P}^k\}$;
        if $W^* > Cost(\phi, C(P_{min}))$ **then** $W^* \leftarrow Cost(\phi, C(P_{min})$ and $C^* \leftarrow C(P_{min})$;
        $S^k \leftarrow PathSelect(\mathcal{P}^k)$;
    end;
    return C^*;
end.

Figure 2: The generic search procedure for identifying profitable multi-exchanges.

Observe that in this procedure, the value of K is a parameter and can be specified by the user. Increasing the value of K may in general improve the quality of local optimal solutions obtained, but our computational investigations show that there are diminishing returns after $K = 4$; hence $K = 4$ is a good value to be used in the search procedure. For another implementation (Implementation 4) of *PathSelect* as discussed in Section 4, we keep 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 is locally optimal. If C^* is nonempty, then it implies that the algorithm found a profitable multi-exchange C^*.

We 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 $\mathcal{P}^k$ and it may contain as many as pn paths. Since computing the cost of k-exchange for each $P \in \mathcal{P}^k$ takes $O(k^2)$ time, we require $O(k^2pn)$ time to find a profitable k-exchange, if any. We shall show in Section 4 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 $O(k^2pn)$.

It is easy to see that if we ignore the time taken by the procedure *PathSelect*, then the bottleneck operation in the generic search procedure is to evaluate the cost $Cost(\phi, C(P))$ of each path $P \in \mathcal{P}^k$. Since $C(P)$ is a k-exchange with respect to the solution ϕ, using (5) we can determine its cost in $O(k^2)$ time. We will next 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 $P = i_1 - i_2 - \ldots - i_k$ 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 $C = i_1 - i_2 - \ldots - i_k - i_1$ except that we do not perform the last move of shifting facility i_k from its current location 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 $i \in N \setminus \{i_1, i_2, \ldots, i_{k-1}\}$. We denote the cost of the path exchange P with respect to the solution ϕ by $Cost(\phi, P)$. Hence,

$$Cost(\phi, P) = z(\phi^P) - z(\phi) = 2 * \sum_{i \in P}\sum_{j=1}^{n} f_{ij}\left(d_{\phi^P(i)\phi^P(j)} - d_{\phi(i)\phi(j)} \right). \tag{7}$$

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:

$$Cost(\phi, C(P)) - Cost(\phi, P) = 2 * c_{i_k i_1}^{\phi}$$
$$+ 2 * \sum_{j \in C} f_{i_k j}\left(\left(d_{\phi(i_1)\phi^C(j)} - d_{\phi(i_k)\phi^P(j)} \right) - \left(d_{\phi(i_1)\phi(j)} - d_{\phi(i_k)\phi(j)} \right) \right). \tag{8}$$

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

$$Cost(\phi, P') - Cost(\phi, P) = 2 * c_{i_k i_{k+1}}^{\phi}$$
$$+ 2 * \sum_{j \in P} f_{i_k j}\left(\left(d_{\phi(i_{k+1})\phi^{P'}(j)} - d_{\phi(i_k)\phi^P(j)} \right) - \left(d_{\phi(i_{k+1})\phi(j)} - d_{\phi(i_k)\phi(j)} \right) \right). \tag{9}$$

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.

Further, when we extend any path P, then the cost of the extended path too can be computed in $O(k)$ time. Thus, the running time of the generic search procedure is $O(K \sum_{k=2}^{K} |\mathcal{P}^k|)$, plus the time taken 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*($\mathcal{P}^k$) differently. The procedure *PathSelect*($\mathcal{P}^k$) accepts as an input a set of paths $\mathcal{P}^k$ and returns a subset S^k of these paths. We describe next several ways in which *PathSelect* can be implemented.

Implementation 1 (All Paths): In this version, we define *PathSelect*($\mathcal{P}^k$) to be $\mathcal{P}^k$ itself; that is, we select all 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 increase exponentially with k and it takes too long to find profitable k-exchanges for $k \geq 6$ even for $n = 25$.

Implementation 2 (Negative Paths): In this version, the subroutine *PathSelect*($\mathcal{P}^k$) returns only those paths which have negative cost; that is, *PathSelect*($\mathcal{P}^k$) = $\{P \in \mathcal{P}^k: Cost(\phi, P) < 0\}$ where ϕ is the current solution. This version is motivated by the intuition that if there is a profitable multi-exchange $C = i_1 - i_2 - \ldots - i_k - i_1$, then there should exist a node in this sequence, say node i_l, so that each of the paths $i_l - i_{l+1}, i_l - i_{l+1} - i_{l+2}, \ldots, i_l - i_{l+1} - i_{l+2} - \ldots - i_{l+k}$ 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.

Implementation 3 (Best αn^2 Paths): In this version, we sort all the paths in $\mathcal{P}^k$ in the non-decreasing order of path costs, and select the first αn^2 paths, where 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 cost are more likely to yield profitable multi-exchanges. The choice of α allows us to strike a right tradeoff between the running time and the solution quality. Higher values of α will 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 8 indicate that $\alpha = 1$ is a good choice considering both the running time and solution quality. We have used max heap data structure to keep αn^2 paths in a stage. Hence if there are pn possible paths (as discussed in Section 3), it takes $pn\log(pn)$ time to store αn^2 best paths in a heap.

Implementation 4 (Best n Paths): In this implementation, we select the best path in $\mathcal{P}^k$ starting at node i for each $1 \leq i \leq n$. 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.

5. THE NEIGHBORHOOD SEARCH ALGORITHM

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

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

Figure 3. The neighborhood Search algorithm for the QAP.

Let us perform the running time analysis of the algorithm. The initial construction of the improvement graph takes $O(n^3)$ time. The time needed by the procedure K-exchange search is $O(K^2 p)$, where p is the maximum number of paths maintained by the procedure during any iteration (see Section 3). For Implementation 3 of *PathSelect*, $p \leq \alpha n^2$ and this procedure requires $O(K^2 n^2)$ time per iteration (that is, per improvement). For Implementation 4 of the *PathSelect*, $p \leq n^2$, and the procedure again takes time $O(K^2 n^2)$. Updating the improvement graph takes $O(n^2 K)$ 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 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 method is not very robust. The QAP in general has an extremely large number of locally optimal solutions even if the size of the neighborhood is very large. Each run produces a locally optimal solution which is a random sample in the solution space of locally optimal solutions. To obtain a robust locally optimal solution, we need to perform several runs of the algorithm and use the best locally optimal solution found in these runs.

6. ACCELERATING THE SEARCH ALGORITHM

In this section, we describe a method to speedup the performance of the generic search algorithm and also its specific implementations. The speedup uses the fact that several paths give 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*, defined as follows:

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

Our generic search algorithm enumerates only valid paths. The following lemma shows that we do not miss any multi-exchanges by maintaining valid paths only.

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

Proof: Consider a multi-exchange $j_1 - j_2 - \dots - j_k$. Let $j_l = \min \{j_h: 1 \leq h \leq k\}$. Now define $i_1 = j_l$, $i_2 = j_{l+1}, \dots, i_k = j_{l+k}$, where all subscript mathematics is modulo $(k+1)$. It follows from the definition of j_l that each of the paths $i_1, i_1 - i_2, i_1 - i_2 - i_3, \dots, i_1 - i_2 - \dots - i_k$ is a valid path. Hence starting at node i_1 we can gradually build $i_1 - i_2 - \dots - i_k$ by maintaining only valid paths, and joining node i_k to node i_1 gives us the desired multi-exchange. ◆

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 $i_1 - i_2 - \dots - i_k$, we compare i_1 with i_{k+1}. If $i_1 \leq i_{k+1}$, we add the arc; otherwise we do not add it. It can be noted that above lemma holds if we enumerate all paths. However, as we keep only αn^2 paths in each stage, there may be the cases when we might miss a profitable multi-exchange. Our experiment shows that loss in missed improvements is well compensated by the gain in time. The computational results presented in Section 8 show that enumerating only valid paths decreases the running time of the generic search algorithm substantially.

7. EXPRESSIONS FOR THE ASYMMETRIC QAP

In the previous sections, we gave expressions for calculating various cost terms for symmetric instances of QAP. In this section, we give expressions for the asymmetric QAP. We state the expressions without proof since their logic is similar to those for the symmetric case.

For the asymmetric case, we will replace the expressions (3)-(9) by the following expressions (3′)-(9′) respectively.

$$Cost(\phi, C) = z(\phi^C) - z(\phi) = \sum_{i \in C} \sum_{j=1}^{n} \left(f_{ij} \left(d_{\phi^C(i)\phi^C(j)} - d_{\phi(i)\phi(j)} \right) + f_{ji} \left(d_{\phi^C(j)\phi^C(i)} - d_{\phi(j)\phi(i)} \right) \right) \qquad (3')$$

$$c_{ij}^{\phi} = z(\phi') - z(\phi) = \sum_{l=1}^{n} \left(f_{il} \left(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)} \right) + f_{li} \left(d_{\phi(l)\phi(j)} - d_{\phi(l)\phi(i)} \right) \right). \qquad (4')$$

$$Cost(\phi, C) = \sum_{(i,j) \in C} c_{ij}^{\phi} - \sum_{i \in C} \sum_{j \in C} \left(f_{ij} \left(d_{\phi^C(i)\phi(j)} - d_{\phi(i)\phi(j)} \right) + f_{ji} \left(d_{\phi(j)\phi^C(i)} - d_{\phi(j)\phi(i)} \right) \right) +$$

$$\sum_{i \in C} \sum_{j \in C} f_{ij} \left(d_{\phi^C(i)\phi^C(j)} - d_{\phi(i)\phi(j)} \right) + \sum_{i \in C} \sum_{j \in C} f_{ji} \left(d_{\phi^C(j)\phi^C(i)} - d_{\phi(j)\phi(i)} \right) \qquad (5')$$

$$c_{ij}^{\phi^C} = c_{ij}^{\phi} - \sum_{l \in C} \Big(f_{il} \big(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)} \big) + f_{li} \big(d_{\phi(l)\phi(j)} - d_{\phi(l)\phi(i)} \big) \Big) +$$

$$\sum_{l \in C} \Big(f_{il} \big(d_{\phi(j)\phi^C(l)} - d_{\phi(i)\phi^C(l)} \big) + f_{li} \big(d_{\phi^C(l)\phi(j)} - d_{\phi^C(l)\phi(i)} \big) \Big) \tag{6'}$$

$$Cost(\phi, P) = z(\phi^P) - z(\phi) = \sum_{i \in P} \sum_{j=1}^{n} \Big(f_{ij} \big(d_{\phi^P(i)\phi^P(j)} - d_{\phi(i)\phi(j)} \big) + f_{ji} \big(d_{\phi^P(j)\phi^P(i)} - d_{\phi(j)\phi(i)} \big) \Big) \tag{7'}$$

$$Cost(\phi, C(P)) - Cost(\phi, P) = c_{i_k i_1}^{\phi} + \sum_{j \in C} f_{i_k j} \Big(\big(d_{\phi(i_1)\phi^C(j)} - d_{\phi(i_k)\phi^P(j)} \big) - \big(d_{\phi(i_1)\phi(j)} - d_{\phi(i_k)\phi(j)} \big) \Big) +$$

$$\sum_{j \in C} f_{j i_k} \Big(\big(d_{\phi^C(j)\phi(i_1)} - d_{\phi^P(j)\phi(i_k)} \big) - \big(d_{\phi(j)\phi(i_1)} - d_{\phi(j)\phi(i_k)} \big) \Big) \tag{8'}$$

$$Cost(\phi, P') - Cost(\phi, P) = c_{i_k i_{k+1}}^{\phi} + \sum_{j \in P} f_{i_k j} \Big(\big(d_{\phi(i_{k+1})\phi^{P'}(j)} - d_{\phi(i_k)\phi^P(j)} \big) - \big(d_{\phi(i_{k+1})\phi(j)} - d_{\phi(i_k)\phi(j)} \big) \Big) +$$

$$\sum_{j \in P} f_{j i_k} \Big(\big(d_{\phi^{P'}(j)\phi(i_{k+1})} - d_{\phi^P(j)\phi(i_k)} \big) - \big(d_{\phi(j)\phi(i_{k+1})} - d_{\phi(j)\phi(i_k)} \big) \Big) \tag{9'}$$

8. COMPUTATIONAL TESTING

In this section, we describe computational results of the neighborhood search algorithms developed by us. We implemented all of our algorithms in C and ran them on IBM SP machine (model RS6000) with a processor speed of 333 MHz. We tested the algorithms on 132 benchmark instances available at the QAPLIB, the library of QAP instances maintained by the Institute of Mathematics, Graz University of Technology (http://www.opt.math.tu-graz.ac.at/qaplib/). Our computational results include analyzing the CPU times taken by our algorithms, quality of the solutions obtained by them as well as understanding the behavior of the VLSN search algorithms.

Neighborhood search algorithms need some feasible solution as the starting solution. We generated random permutations of n numbers and used them as starting solutions. Further we implemented a multi-start version of the neighborhood search algorithm, where we apply the neighborhood search algorithm multiple times with different starting solutions, called different *runs,* and select the best solution found in these runs. Number of runs depend 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 the algorithm runs very slowly even when we go up to k-exchanges with $k = 6$. For example, to solve a QAP with $n = 42$ (instance sko42), each run of this implementation takes about 8 seconds for $k = 4$ whereas Implementation 3 takes only 0.025 second per run. Additional preliminary tests yielded that this

implementation is not as competitive as other implementations and we decided not to perform a thorough testing of the algorithm.

In the second implementation of the VLSN search algorithm, we maintain only those paths which 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 non-linearity in the cost structure. Our computational testing revealed that maintaining only negative cost paths is not a good heuristic to enumerate negative cost cycles. Thus, we did not performed a thorough testing of this implementation.

Our preliminary testing revealed that Implementation 3 and 4 exhibited the best overall behavior and deserved a thorough testing. The following details of implementation 3 are worth mentioning. Recall from Section 4 that we keep only αn^2 best paths in $\mathcal{P}^k$. We used the Max Heap data structure (Cormen et al. [2001]) to store these paths. We found that $\alpha = 1$ gives fairly good results and hence we used this value. In addition, we used only those paths whose path cost is not more than 0.5% of the best objective function value of the QAP found so far. We found that using higher cost paths rarely leads to negative cost cycles. Finally, when we examined paths in $\mathcal{P}^k$ to enumerate cycles of length k and find several negative cost cycles, we use the least cost negative cycle to obtain the next solution. As far as Implementation 4 is concerned, we implemented it in the straightforward fashion but before enumerating paths, we eliminate all negative cycles of length 2 by performing 2-exchanges.

Accuracy of the Solution

We applied Implementation 3 and 4 to the 132 benchmark instances in QAPLIB, of these 98 were instances of symmetric QAP and the remaining were for the asymmetric case. 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 1 hour for $n \leq 40$ and for 2 hours for $n > 40$. The running times for the asymmetric instances were 1.5 hours for $n \leq 40$ and for 3 hours for $n > 40$. Figures 4 and 5, respectively, give 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 BestGap, AvgGap, nRuns, %Best, respectively, give the percent deviation of the best solution found in all runs with respect to the best known solution, average deviation over solutions found in all runs, the number of runs, and the percentage of the solutions found which were best known solutions. We can derive the following conclusions from these tables.

- Implementation 3 exhibited the best overall performance. It obtained the best-known solutions in 74 out of 98 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.

16

- Implementation 3 is found to exhibit superior performance compared to 2OPT in terms of the gap of the best solution found by algorithm with the best-known solution. For 25 symmetric instances, Implementation 3 obtained better solutions than 2OPT, and for only 2 symmetric instances 2OPT obtained better solutions than Implementation 3. Similarly, for 10 asymmetric instances implementation 3 obtained better solutions than 2OPT, and for only 1 asymmetric instance 2OPT obtained better solution than Implementation 3.

- Implementation 3 is also found to be better than 2OPT in terms of the average gap and the frequency of finding best-known solution. The average of AvgGap of Implementation 3 was 7.6%, whereas this number for 2OPT was 11.05% for the symmetric instances and these numbers were 6.42% and 7.49% respectively for the asymmetric instances. Finally, whereas Implementation 3 found best-known solution with an average frequency of 17.13% in symmetric case, this number for 2OPT was 11.85% in symmetric case. For asymmetric case, Implementation 3 found best-known solution with an average frequency of 1.97%, whereas this number for 2OPT was 0.46%.

- Implementation 4 also exhibited superior performance with respect to 2OPT, but its overall performance was worse than Implementation 3. Implementation 4 runs very fast and it terminates in a fraction of second for most problem sizes, but the solutions obtained using this method are not as robust as those obtained using Implementation 3.

Above results seem to suggest that very large-scale neighborhood is overall more effective than the traditional 2-exchange neighborhood. When both the algorithms are run for the same time, the 2OPT performs many more runs but still the best solution found is, on the average, not as good as found by VLSN search in lesser number of runs. Hence the extra time taken by VLSN search algorithm is more than justified by the better quality of the solutions obtained.

We will now describe some computational investigations we performed to understand the behavior of our implementations.

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 paths maintained of a given length. The larger the cycle length and the number of paths maintained, greater is the neighborhood, more is the running time, and better is the quality of the solution obtained (in general). Hence 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, sko100f, and applied 100 runs of Implementation 3 with cycle lengths varying from 2 to 7 and noted the average running time taken by the algorithm (per run) and the average gap (per run). We kept the number of paths maintained by the algorithm as fixed at n^2. Figure 6 plots these two values as a function of cycle length. It is easy to see that the average gap decreases significantly with the increase in cycle length until cycle length is 4, and 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. We think that the cycle length of 4 strikes a right balance between the solution accuracy and solution time and hence we used this value in the computational results presented earlier.
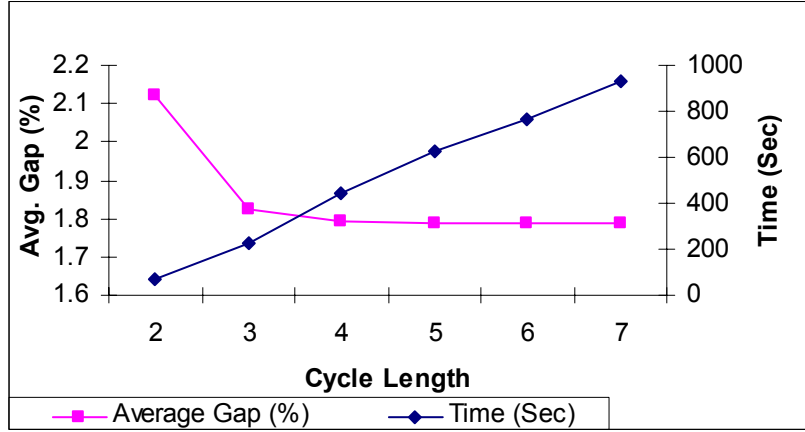
Figure 6: Effect of cycle length on time taken and solution quality for 100 runs on problem sko100a-f.

Our second experiment was similar to the first experiment but we varied the number of paths maintained by the algorithm while keeping the cycle length fixed at 4. Figure 7 gives a plot of the average gap and average time per run when we performed 100 runs of Implementation 3 on the six problems sko100a-f. We observe that the solution accuracy gradually improves as the number of paths increase as well as the running time of algorithm increases linearly with the number of paths maintained. We believe that maintaining n^2 paths is a good compromise between solution quality and solution time and we used this value in our experiments.
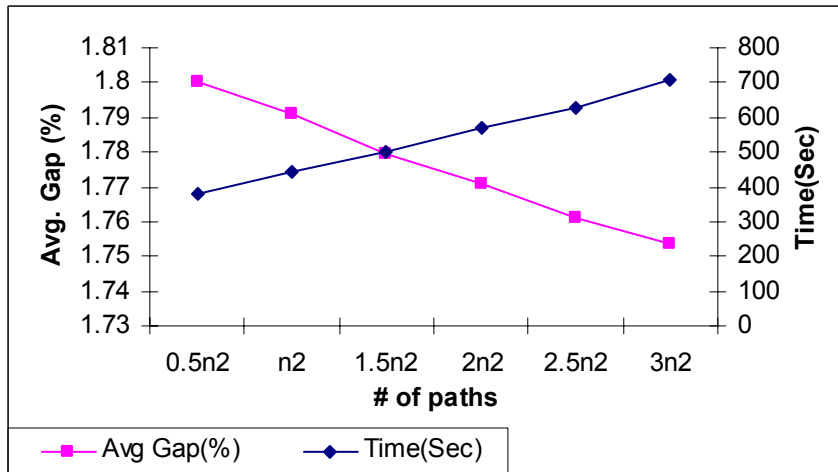
Figure 7: Effect of number of paths in each stage on time taken and solution quality for 100 runs on problem sko100a-f.

In another experiment, we counted the number of improvement iterations with cycle length 2, 3 and 4. Recall that our algorithm performs a 3-exchange when it fails to find 2-

exchange, and performs a 4-exchange when it fails to find a 3-exchange. The table shown in Figure 8 gives these values for 10 benchmark instances on which we apply 100 runs of Implementation 3. We observe that there are many more iterations with 2-exchanges compared to 3-exchanges, and many more 3-exchanges compared to 4-exchanges.

Problem	# of iterations of Cycle Length		
	2	3	4
chr22a	1614	151	49
kra30a	2283	123	30
kra30b	2306	125	32
nug30	2580	104	42
ste36a	3537	142	44
tho40	3839	124	35
wil50	5298	81	38
sko42	4246	150	64
sko100a	13232	270	70
tai100a	7267	274	75

Figure 8: Number of iterations with different implemented cycle length.

Effect of the Speedup Technique

The reader may recall from Section 6 that we used a speedup technique to reduce redundant enumeration of cycles. In this technique, we maintain only those valid paths $i_1, i_2, ..., i_k$ for which $i_k > i_1$. Lemma 2 showed that we would 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. The table shown in Figure 9 gives these values for 10 benchmark instances. We applied 100 runs on each benchmark instances and noted the average values. We observe that speedup technique decreases the running time substantially but also worsens the solution quality. We believe that overall it is advantageous to use the speedup technique since the saved time can be used to perform more runs of the algorithm and improve the overall performance of the algorithm.

	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	1191

Figure 9: Effect of accelerated path enumeration scheme.

9. CONCLUSIONS

In this paper, we develop a very large-scale neighborhood structure for the QAP. We show that using the concept of improvement graph, we can easily and quickly enumerate multi-exchange neighbors of a given solution. We develop a generic search procedure to enumerate and evaluate neighbors and propose several specific implementations of the generic procedure. We perform extensive computational investigations of our implementations and have found concerning evidence that multi-exchange neighborhoods add value over 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 more on neighborhood structure and less on specific implementations. Further possibilities for improvement could possibly be obtained using ideas from tabu search (Glover and Laguna [1997]). We leave it as a topic of future research. Neighborhood search algorithms have also been used in genetic algorithms to improve the quality of the individuals in the population (Ahuja, Orlin, and Tiwari [2000], and Drezner [2001]). Our neighborhood structure may be useful in these genetic algorithms too.

SN	Name	n	BKS	2OPT				Implementation 3 (Best αn^2 paths)				Implementation 4 (Best n paths)			
				BestGap	AvgGap	nRuns	%Best	BestGap	AvgGap	nRuns	%Best	BestGap	AvgGap	nRuns	%Best
1	Chr12a	12	9552	0	45.76	9,436,220	0.89	0	31.21	2,824,741	2.34	0	28.98	3,807,890	1.45
2	Chr12b	12	9742	0	52.78	8,402,658	6.42	0	45.37	3,002,710	8.26	0	44.38	4,572,662	8.64
3	Chr12c	12	11156	0	34.92	9,823,150	0.32	0	24.95	2,957,361	0.76	0	23.82	3,907,786	0.52
4	Chr15a	15	9896	0	50.10	4,704,652	0.07	0	34.06	1,420,416	0.38	0	32.43	2,021,011	0.41
5	Chr15b	15	7990	0	60.11	4,468,591	0.17	0	47.09	1,537,817	0.25	0	46.64	2,266,858	0.37
6	Chr15c	15	9504	0	61.61	4,920,512	0.04	0	43.86	1,395,482	0.09	0	41.79	2,015,713	0.19
7	Chr18a	18	11098	0	66.83	2,730,190	0.01	0	50.01	779,653	0.17	0	49.76	1,285,044	0.06
8	Chr18b	18	1534	0	14.91	3,061,778	0.57	0	8.53	749,002	2.84	0	14.34	2,699,902	0.92
9	Chr20a	20	2192	0	47.81	2,102,113	0.00	0	33.83	531,198	0.02	0	43.11	1,537,470	0.00
10	Chr20b	20	2298	0	41.48	2,248,147	0.00	0	27.59	545,328	0.00	0	33.18	1,304,395	0.00
11	Chr20c	20	14142	0	83.48	1,638,126	0.08	0	63.75	537,304	0.20	0	68.17	1,004,525	0.23
12	Chr22a	22	6156	0	13.75	1,409,341	0.00	0	10.03	321,233	0.02	0	12.73	1,094,969	0.00
13	Chr22b	22	6194	0.581	13.98	1,524,784	0.00	0	9.55	306,967	0.00	0	11.86	935,420	0.00
14	Chr25a	25	3796	0	57.88	921,946	0.00	0	44.08	261,426	0.00	0	52.94	700,777	0.00
15	Els19	19	17212548	0	25.63	1,703,953	1.91	0	14.47	298,488	23.78	0	24.86	1,501,674	1.91
16	Esc16a	16	68	0	3.54	6,915,006	35.00	0	0.36	1,187,393	94.63	0	3.54	6,849,543	35.00
17	Esc16b	16	292	0	0.01	9,060,254	98.44	0	0.00	1,063,243	100.00	0	0.01	8,988,691	98.44
18	Esc16c	16	160	0	1.15	5,753,649	53.58	0	0.25	1,140,441	84.19	0	1.15	5,702,568	53.58
19	Esc16d	16	16	0	9.18	7,089,066	43.13	0	0.72	1,165,768	94.29	0	9.18	7,026,224	43.13
20	Esc16e	16	28	0	10.12	8,572,391	20.17	0	3.25	1,165,223	60.37	0	10.12	8,502,350	20.17
21	Esc16f	16	0	0	0.00	35,611,664	100.00	0	0.00	4,918,942	100.00	0	0.00	35,420,396	100.00
22	Esc16g	16	26	0	7.86	7,348,488	44.39	0	0.23	1,124,152	97.03	0	7.86	7,287,152	44.39
23	Esc16h	16	996	0	0.00	7,922,830	100.00	0	0.00	1,051,718	100.00	0	0.00	7,857,457	100.00
24	Esc16i	16	14	0	1.09	8,107,025	96.53	0	0.00	1,712,511	100.00	0	1.09	8,041,288	96.53
25	Esc16j	16	8	0	15.49	9,128,130	54.33	0	1.28	940,748	94.90	0	15.47	9,037,786	54.34
26	Esc32a	32	130	0	23.16	591,484	0.00	0	13.48	105,197	0.13	0	23.16	589,319	0.00
27	Esc32b	32	168	0	28.83	573,480	0.49	0	15.60	119,828	3.89	0	28.83	571,874	0.49
28	Esc32c	32	642	0	0.40	806,970	82.10	0	0.01	116,680	99.87	0	0.40	804,161	82.10
29	Esc32d	32	200	0	6.42	789,986	4.49	0	3.57	115,082	21.64	0	6.42	787,713	4.49
30	Esc32e	32	2	0	0.00	1,862,801	100.00	0	0.00	118,660	100.00	0	0.00	1,858,335	100.00
31	Esc32f	32	2	0	0.00	1,861,132	100.00	0	0.00	118,654	100.00	0	0.00	1,858,965	100.00
32	Esc32g	32	6	0	0.64	1,868,071	98.08	0	0.00	116,371	100.00	0	0.64	1,865,820	98.08
33	Esc32h	32	438	0	3.93	732,021	1.55	0	2.00	82,942	9.08	0	3.93	728,741	1.55
34	Esc64a	64	116	0	1.48	302,102	48.88	0	0.14	21,571	95.69	0	1.48	301,986	48.88

Figure 4: Computational results for symmetric instances. **Contd ...**

SN	Name	n	BKS	2OPT				Implementation 3 (Best αn^2 paths)				Implementation 4 (Best n paths)			
				BestGap	AvgGap	nRuns	%Best	BestGap	AvgGap	nRuns	%Best	BestGap	AvgGap	nRuns	%Best
35	Esc128	128	64	0	13.10	38,994	8.27	0	5.61	1,850	33.89	0	13.09	39,101	8.26
36	Had12	12	1652	0	1.31	8,903,617	4.97	0	0.72	1,571,863	18.01	0	1.20	6,737,857	6.69
37	Had14	14	2724	0	0.87	4,681,890	13.68	0	0.48	979,815	32.92	0	0.82	3,890,031	14.49
38	Had16	16	3720	0	0.90	3,046,483	13.75	0	0.48	674,610	26.19	0	0.87	2,658,712	13.81
39	Had18	18	5358	0	1.10	2,224,949	2.66	0	0.78	407,788	5.32	0	1.07	2,016,232	2.88
40	Had20	20	6922	0	1.19	1,562,491	2.27	0	0.84	289,189	6.76	0	1.17	1,435,751	2.29
41	Kra30a	30	88900	0	7.70	508,286	0.02	0	5.99	82,846	0.20	0	7.70	507,363	0.02
42	Kra30b	30	91420	0	5.76	505,339	0.00	0	4.13	83,349	0.04	0	5.76	504,516	0.00
43	Nug12	12	578	0	5.29	10,480,616	1.35	0	3.48	2,387,442	6.50	0	4.85	6,507,051	2.36
44	Nug14	14	1014	0	5.02	6,045,903	0.44	0	3.42	1,096,624	0.95	0	4.70	3,920,230	0.86
45	Nug15	15	1150	0	4.46	4,738,236	1.46	0	2.79	1,005,809	4.94	0	4.02	3,224,136	1.97
46	Nug16a	16	1610	0	4.86	3,835,499	0.28	0	3.43	698,454	1.61	0	4.56	2,729,922	0.53
47	Nug16b	16	1240	0	5.40	3,854,426	3.12	0	3.64	756,449	6.98	0	4.93	2,615,318	4.06
48	Nug17	17	1732	0	4.24	3,103,273	0.08	0	2.73	553,263	1.08	0	4.01	2,283,745	0.14
49	Nug18	18	1930	0	4.44	2,633,371	0.17	0	3.09	467,170	0.85	0	4.15	1,876,454	0.25
50	Nug20	20	2570	0	4.19	1,856,594	0.22	0	3.04	342,835	0.67	0	3.97	1,386,350	0.26
51	Nug21	21	2438	0	4.58	1,481,678	0.11	0	3.14	283,327	0.29	0	4.35	1,188,565	0.14
52	Nug22	22	3596	0	3.72	1,195,612	0.50	0	2.71	244,287	1.37	0	3.59	1,015,230	0.53
53	Nug24	24	3488	0	4.67	967,896	0.10	0	3.31	181,818	0.70	0	4.45	787,586	0.26
54	Nug25	25	3744	0	3.90	842,354	0.05	0	2.64	160,132	0.42	0	3.80	726,571	0.06
55	Nug27	27	5234	0	4.33	626,472	0.04	0	3.27	115,644	0.40	0	4.16	527,571	0.11
56	Nug28	28	5166	0	4.51	579,278	0.01	0	3.30	100,579	0.19	0	4.32	481,522	0.02
57	Nug30	30	6124	0	4.19	453,552	0.01	0	3.06	86,179	0.03	0	4.10	398,408	0.01
58	Rou12	12	235528	0	5.64	10,263,930	0.69	0	4.25	2,348,174	1.83	0	4.65	4,692,485	1.09
59	Rou15	15	354210	0	6.90	5,230,145	0.37	0	5.26	1,081,279	0.93	0	6.06	2,631,306	0.57
60	Rou20	20	725522	0	4.86	2,114,540	0.01	0	3.34	344,848	0.02	0	4.36	1,256,472	0.01
61	Scr12	12	31410	0	7.03	9,411,221	5.05	0	4.67	2,337,375	11.82	0	4.82	4,371,642	16.64
62	Scr15	15	51140	0	10.24	4,273,122	1.76	0	7.66	1,070,211	6.80	0	7.65	2,114,390	5.51
63	Scr20	20	110030	0	9.86	1,709,033	0.06	0	6.22	378,005	0.21	0	6.52	896,590	0.18
64	Sko42	42	15812	0.101	3.53	294,068	0.00	0	2.73	53,925	0.01	0.101	3.48	270,262	0.00
65	Sko49	49	23386	0.162	3.08	176,442	0.00	0.154	2.44	33,002	0.00	0.162	3.03	165,944	0.00
66	Sko56	56	34458	0.430	2.96	108,971	0.00	0.163	2.39	20,623	0.00	0.261	2.92	103,033	0.00

Figure 4 (contd.): Computational results for symmetric instances.   Contd ...

SN	Name	n	BKS	2OPT				Implementation 3 (Best αn^2 paths)				Implementation 4 (Best n paths)			
				BestGap	AvgGap	nRuns	%Best	BestGap	AvgGap	nRuns	%Best	BestGap	AvgGap	nRuns	%Best
67	Sko64	64	48498	0.421	2.70	68,901	0.00	0.293	2.20	12,162	0.00	0.400	2.68	66,537	0.00
68	Sko72	72	66256	0.420	2.65	45,817	0.00	0.546	2.21	7,832	0.00	0.420	2.64	44,570	0.00
69	Sko81	81	90998	0.510	2.27	31,087	0.00	0.440	1.91	5,273	0.00	0.510	2.26	30,521	0.00
70	Sko90	90	115534	0.535	2.23	21,473	0.00	0.460	1.89	3,425	0.00	0.535	2.22	21,126	0.00
71	Sko100a	100	152002	0.617	2.08	14,955	0.00	0.529	1.74	2,180	0.00	0.617	2.07	14,750	0.00
72	Sko100b	100	153890	0.516	2.02	15,110	0.00	0.377	1.71	2,222	0.00	0.516	2.02	14,908	0.00
73	Sko100c	100	147862	0.580	2.30	14,843	0.00	0.511	1.96	2,182	0.00	0.580	2.29	14,633	0.00
74	Sko100d	100	149576	0.646	2.07	15,170	0.00	0.513	1.74	2,179	0.00	0.646	2.07	15,038	0.00
75	Sko100e	100	149150	0.581	2.31	14,820	0.00	0.457	1.94	2,187	0.00	0.581	2.30	14,615	0.00
76	Sko100f	100	149036	0.780	2.03	15,299	0.00	0.573	1.71	2,215	0.00	0.666	2.02	15,083	0.00
77	Ste36a	36	9526	0.252	12.02	238,239	0.00	0	9.07	44,939	0.01	0.252	12.02	239,827	0.00
78	Ste36b	36	15852	0	21.46	213,196	0.01	0	15.95	47,338	0.18	0	21.46	214,163	0.01
79	Ste36c	36	8239.11	0.132	9.46	226,567	0.00	0	7.24	45,428	0.00	0.132	9.46	227,512	0.00
80	Tai12a	12	224416	0	8.93	10,073,866	2.12	0	6.88	1,715,403	4.79	0	7.86	4,651,037	3.09
81	Tai15a	15	388214	0	4.81	5,387,038	0.12	0	3.41	986,214	0.40	0	4.07	2,661,777	0.15
82	Tai17a	17	491812	0	5.69	3,620,256	0.05	0	4.22	627,478	0.21	0	4.89	1,846,451	0.17
83	Tai20a	20	703482	0	6.03	2,214,371	0.00	0	4.37	340,557	0.02	0	5.11	1,147,683	0.01
84	Tai25a	25	1167256	0	5.57	1,109,310	0.00	0	4.08	173,744	0.00	0	4.76	609,159	0.00
85	Tai30a	30	1818146	0.456	5.06	615,703	0.00	0.532	3.84	78,638	0.00	0	4.47	371,344	0.00
86	Tai35a	35	2422002	1.083	5.10	387,154	0.00	0.687	3.72	46,236	0.00	1.075	4.47	237,949	0.00
87	Tai40a	40	3139370	1.358	5.05	254,304	0.00	0.906	3.68	28,821	0.00	1.357	4.56	169,185	0.00
88	Tai50a	50	4941410	1.897	4.97	253,480	0.00	1.437	3.71	26,559	0.00	1.863	4.46	170,079	0.00
89	Tai60a	60	7208572	2.177	4.72	138,643	0.00	1.658	3.54	13,200	0.00	1.902	4.20	91,441	0.00
90	Tai64c	64	1855928	0	0.44	416,518	6.09	0	0.42	27,985	6.19	0	0.44	418,391	6.09
91	Tai80a	80	13557864	2.037	3.76	56,268	0.00	1.558	2.78	4,484	0.00	1.713	3.22	36,122	0.00
92	Tai100a	100	21125314	1.969	3.42	27,534	0.00	1.608	2.49	1,789	0.00	1.707	3.03	19,184	0.00
93	Tai256c	256	44759294	0.175	0.43	3,526	0.00	0.175	0.41	168	0.00	0.175	0.43	3,392	0.00
94	Tho30	30	149936	0	4.85	432,914	0.01	0	3.72	86,314	0.03	0	4.60	336,287	0.01
95	Tho40	40	240516	0.341	4.67	174,638	0.00	0.091	3.70	35,056	0.00	0.043	4.46	142,400	0.00
96	Tho150	150	8133484	0.877	2.42	3,578	0.00	0.814	2.12	469	0.00	0.900	2.39	2,589	0.00
97	Wil50	50	48816	0.213	1.66	18,331	0.00	0.086	1.37	30,101	0.00	0.123	1.65	149,664	0.00
98	Wil100	100	273038	0.363	1.10	15,005	0.00	0.347	0.95	2,031	0.00	0.363	1.10	14,921	0.00

Figure 4 (contd.): Computational results for symmetric instances.

SN	Name	n	BKS	2OPT				Implementation 3 (Best αn^2 paths)				Implementation 4 (Best n paths)			
				BestGap	AvgGap	nRuns	%Best	BestGap	AvgGap	nRuns	%Best	BestGap	AvgGap	nRuns	%Best
1	bur26a	26	5426670	0	0.32	114,672	0.32	0	0.24	85,634	1.11	0	0.32	114,606	0.32
2	bur26b	26	3817852	0	0.41	122,308	0.29	0	0.30	87,293	0.58	0	0.41	122,247	0.29
3	bur26c	26	5426795	0	0.41	112,625	0.31	0	0.26	82,912	1.05	0	0.41	112,556	0.31
4	bur26d	26	3821225	0	0.47	119,809	0.28	0	0.29	87,760	0.54	0	0.47	119,743	0.28
5	bur26e	26	5386879	0	0.37	110,953	0.39	0	0.20	83,787	2.65	0	0.37	110,890	0.39
6	bur26f	26	3782044	0	0.46	119,957	0.66	0	0.24	87,391	3.07	0	0.46	119,891	0.66
7	bur26g	26	10117172	0	0.36	109,920	0.47	0	0.25	83,347	2.44	0	0.36	109,829	0.47
8	bur26h	26	7098658	0	0.46	116,744	0.95	0	0.35	93,540	4.50	0	0.46	116,677	0.95
9	lipa20a	20	3683	0	2.84	386,856	0.37	0	2.52	238,081	1.37	0	2.80	285,714	0.50
10	lipa20b	20	27076	0	15.22	364,584	4.27	0	12.66	277,407	12.94	0	14.80	233,970	4.96
11	lipa30a	30	13178	0	2.02	104,987	0.03	0	1.83	54,075	0.24	0	2.02	104,939	0.03
12	lipa30b	30	151426	0	16.79	102,598	1.88	0	14.96	56,856	7.34	0	15.94	61,550	4.03
13	lipa40a	40	31538	0.907	1.52	43,018	0.00	0	1.36	18,082	0.01	0.907	1.51	36,566	0.00
14	lipa40b	40	476581	0	18.64	41,186	1.26	0	16.90	19,034	5.80	0	18.23	29,478	2.03
15	lipa50a	50	62093	0.892	1.31	42,978	0.00	0.849	1.17	15,372	0.00	0.892	1.30	37,043	0.00
16	lipa50b	50	1210244	0	18.67	41,633	0.46	0	17.52	16,644	2.33	0	18.40	30,964	0.70
17	lipa60a	60	107218	0.844	1.11	24,248	0.00	0.787	0.99	7,166	0.00	0.844	1.10	21,339	0.00
18	lipa60b	60	2520135	0	20.08	24,046	0.09	0	19.22	7,938	0.42	0	19.71	16,725	0.22
19	lipa70a	70	169755	0.775	0.96	15,016	0.00	0.725	0.86	4,056	0.00	0.773	0.95	13,187	0.00
20	lipa70b	70	4603200	0	20.80	14,644	0.05	0	19.94	4,358	0.53	0	20.37	9,767	0.17
21	lipa80a	80	253195	0.678	0.85	10,026	0.00	0.628	0.75	2,335	0.00	0.678	0.83	8,510	0.00
22	lipa80b	80	7763962	0	21.69	9,701	0.03	0	20.92	2,451	0.08	0	21.23	6,214	0.06
23	lipa90a	90	360630	0.636	0.78	6,836	0.00	0.581	0.69	1,412	0.00	0.636	0.77	6,231	0.00
24	tai12b	12	39464925	0	11.25	1,619,258	3.03	0	8.94	1,682,704	12.55	0	10.35	953,511	3.56
25	tai20b	20	122455319	0	17.45	266,289	0.60	0	14.22	207,118	6.30	0	15.13	183,113	0.88
26	tai25b	25	344355646	0	16.24	121,862	0.03	0	12.10	83,619	0.59	0	15.53	89,529	0.08
27	tai30b	30	637117113	0	13.72	65,700	0.00	0	9.06	38,047	0.13	0	12.65	28,945	0.00
28	tai35b	35	283315445	0	8.72	40,095	0.00	0	6.47	23,213	0.03	0	8.12	31,932	0.01
29	tai40b	40	637250948	0	10.59	24,696	0.00	0	8.29	13,527	0.24	0	10.18	19,742	0.01
30	tai50b	50	458821517	0.073	7.24	23,734	0.00	0.058	5.73	11,315	0.00	0.073	7.10	20,253	0.00
31	tai60b	60	608215054	0.008	7.99	12,713	0.00	0.038	6.16	4,863	0.00	0.089	7.75	10,853	0.00
32	tai80b	80	818415043	1.594	6.11	4,988	0.00	0.844	5.27	1,597	0.00	1.594	6.01	4,563	0.00
33	tai100b	100	1185996137	0.875	5.31	2,298	0.00	0.653	4.43	563	0.00	0.762	5.20	2,116	0.00
34	tai150b	150	498896643	1.473	3.49	624	0.00	1.467	3.10	92	0.00	1.473	3.44	584	0.00

Figure 5: Computational results for asymmetric instances.

REFERENCES

Aarts, E. H. L., and J. Korst. *Simulated Annealing and Boltzman Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. Wiley, Chichester, 1989.

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

Ahuja, R. K., O. Ergun, J. B. Orlin, and A.P. Punnen. 2002. A survey of very large scale neighborhood search techniques. To appear in *Discrete Applied Mathematics*.

Ahuja, R. K., J. B. Orlin, and A. Tiwari. 2000. A greedy genetic algorithm for the quadratic assignment problem. *Computers and Operations Research* **27**, 917-934.

Ahuja, R. K., J. B. Orlin, D. Sharma. 2001a. Multi-exchange neighborhood search algorithms for the capacitated minimum spanning tree problem. *Mathematical 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. Submitted to *Operations Research Letters*.

Ahuja, R. K., J. B. Orlin, D. Sharma. 2001c. A very large-scale neighborhood search algorithm for the combined through-fleet assignment model. Submitted to *INFORMS Journal on Computing*.

Armour, G. C., and E. S. Buffa. 1963. Heuristic algorithm and simulation approach to relative location of facilities. *Management Science* **9**, 294-309.

Bazara, M. S., and M. D. Sherali. 1980. Benders' partitioning scheme applied to a new formulation of the quadratic assignment problem. *Naval Research Logistics Quarterly* **27**, 29-41.

Bean, J. C. 1994. Genetic algorithms and random keys for sequencing and optimization. *ORSA Journal on Computing* **6**, 154-160.

Buffa, E. S., G. C. Armour, and T. E. Vollmann. 1964. Allocating facilities with CRAFT. *Harvard Business Review* **42**, 136-158.

Burkard, R. E. 1991. Location with spatial interactions: The quadratic assignment problem. *Discrete Location Theory* (Eds. P. B. Mirchandani and R. L. Francis), John Wiley.

Burkard, R. E., and T. Bonniger. 1983. A heuristic for quadratic boolean programs with applications to quadratic assignment problems. *European Journal of Operations Research* **13**, 374-386.

Burkard, R. E., E. Cela, P. M. Pardalos, and L. S. Pitsoulis. 1998. The quadratic assignment problem. In Handbook of Combinatorial Optimization **3**, Edited by D. Z. Zhu and P. M. Pardalos, Kluwer, 241-337.

Burkard, R. E., S. E. Karisch, and F. Rendl. 1997. QAPLIB - A quadratic assignment program library. *Journal of Global Optimization* **10**, 391-403.

Cela, E. 1998. The Quadratic Assignment Problem – Theory and Algorithms, Kluwer Academic Publishers.

Christofides, N., and E. Benavent. 1989. An exact algorithm for the quadratic assignment problem. *Operations Research* **37**, 760-768.

Cormen T. H., C. E. Leiserson, R. L. Rivest, C. Stein. 2001. Introduction to Algorithms, Second Edition. The MIT Press.

Davis. L. 1991. *Handbook of Genetic Algorithms*. Van Nostrand, New York.

Deineko V. G., and G.J. Woeginger. 2000. A study of exponential neighborhoods for the Travelling Salesman Problem and for the Quadratic Assignment Problem. *Mathematical Programming* **87(3),** 519-542.

Drezner, Z. 2001. A new genetic algorithm for the quadratic assignment problem. Technical Report, College of Business and Economics, California State University-Fullerton, Fullerton, CA 92834.

Ergun, O. 2001. New Neighborhood Search Algorithms Based on Exponentially Large Neighborhoods. *Operations Research Center, MIT*

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

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

Ibaraki, T., M. Kubo, T. Masuda, T. Uno, and M. Yagiura. 2002. Effective local search algorithms for the vehicle routing problem with general time window constraints. *Working Paper*.

Lawler, E. L. 1963. The quadratic assignment problem. *Management Science* **9**, 586-599.

Li, Y., P. M. Pardalos, and M. G. C. Resende. 1994. A greedy randomized adaptive search procedure for the quadratic assignment problem. In "Quadratic Assignment and Related Problems", Edited by P. M. Pardalos and H. Wolkowicz, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science,* Americal Mathematical Society, 237-261.

Malucelli, F. 1993. *Quadratic Assignment Problems: Solution Methods and Applications.* Unpublished Doctoral Dissertation, Departimento di Informatica, Universita di Pisa, Italy.

Maniezzo, V., A. Colorni, and M. Dorigo. 1994. The ant system applied to the quadratic assignment problem. Tech. Rep. IRIDIA/94-28 , Université Libre de Bruxelles, Belgium.

Muller-Merbach, H. 1970. *Optimale Reihenfolgen*. Springer Verlag, Berlin, 158-171.

Nugent, C. E., T. E. Vollman, and J. Ruml. 1968. An experimental comparison of techniques for the assignment of facilities to locations. *Operations Research* **16**, 150-173.

Pardalos, P. M., and J. Crouse. 1989. A parallel algorithm for the quadratic assignment problem. *Proceedings of the Supercomputing 1989 Conference*, ACM Press, pp. 351-360.

Pardalos, P.M., F. Rendl, and H. Wolkowicz. 1994. The quadratic assignment problem. In "*Quadratic Assignment and Related Problems*", edited by P. M. Pardalos and H. Wolkowicz, DIMACS Series, American Mathematical Society, pp. 1-42.

Resende, M. G. C., P. M. Pardalos, and Y. Li. 1994. Fortran subroutines for approximate solution of dense quadratic assignment problems using GRASP. *ACM Transactions on Mathematical Software* **22**, 104-118.

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

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

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

Tate, D. E. and A. E. Smith. 1985. A genetic approach to the quadratic assignment problem. *Computers and Operations Research* **22**, 73-83.

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

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

West, D. H. 1983. Algorithm 608 : Approximate solution of the quadratic assignment problem. *ACM Transactions on Mathematical Software* **9**, 461-466.

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