LA-UR- 09-00056

Approved for public release;
distribution is unlimited.

Title:	Large Neighborhood Search for the Double Traveling Salesman Problem with Multiple Stacks
Author(s):	Russell Bent Pascal Van Hentenryck
Intended for:	Submission to the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2009).

Los Alamos
NATIONAL LABORATORY
EST.1943

Form 836 (7/06)

Large Neighborhood Search for the Double Traveling Salesman Problem with Multiple Stacks

Russell Bent[1] and Pascal Van Hentenryck[2]

[1] Los Alamos National Laboratory, Los Alamos NM 87545, USA
[2] Brown University, Providence RI 02912, USA

Abstract. This paper considers a complex real-life short-haul/long haul pickup and delivery application. The problem can be modeled as double traveling salesman problem (TSP) in which the pickups and the deliveries happen in the first and second TSPs respectively. Moreover, the application features multiple stacks in which the items must be stored and the pickups and deliveries must take place in reserve (LIFO) order for each stack. The goal is to minimize the total travel time satisfying these constraints. This paper presents a large neighborhood search (LNS) algorithm which improves the best-known results on 65% of the available instances and is always within 2% of the best-known solutions.

1 Introduction

Vehicle routing problems (VRP) and traveling salesman problems (TSP) have received considerable attention in the last decades due to the crucial role they play in many supply-chain and logistics operations. These problems are often approached by meta-heuristics, since problems with as few as 100 customers are often beyond the scope of state-of-the-art systematic search algorithms. Recent work on the VRP and TSP has produced significant improvements in solution quality and execution time, often by combining several approaches or heuristics, and has focused on problems with traditional constraints such as capacities and time windows. Comparatively little research has been devoted to side constraints that, while less prevalent in the literature, are ubiquitous in practice.

This paper considers the Double Traveling Salesman Problem with Multiple Stacks (DTSPMS) which was introduced recently in [13]. Customers in the DTSPMS are divided into pickup and delivery pairs. Given such a pair (p, d), a routing must schedule the pickup customer p before the delivery customer d. In addition, the vehicle must visit all the pickup customers first, returns to a depot, and then visits all the delivery customers, which explains the the double TSP nature of the problem. Finally, the vehicle has a fixed number of rows/stacks which are used to store the picked up items and the deliveries must take place in the reverse order in which they were loaded into a row, giving rise to the multi-stack nature of the problem. The objective is to minimize the total travel cost incurred by the vehicle.

The DTSPMS was introduced in [13] to model a real-life short-haul/long haul combined pickup and delivery applications. Goods are picked up in one local area and delivered "en mass" to a long-haul operation (e.g., a freight train) where, due to labor, time, union rules, and fragility, the goods cannot be re-packed and must be shipped as is. Upon reaching the long-haul destination, the goods are delivered in reverse order in which they were packed. Once again, due to labor and time-intensive costs, the vehicle may only deliver goods that are not blocked by other goods.

The difficulty in the DTSPMS lies in the side-constraints and the stack assignment variables, which complicate the neighborhoods and invalidates many of the traditional VRP moves [14]. Indeed, the stack in which a pickup customer is placed severely restricts the order in which a delivery customer may be served. The pickup and delivery constraints are more standard and arise in many applications such as dial-a-ride problems, airline scheduling, bus routing, tractor-trailer problems, helicopter support of offshore oil field platforms, and logistics and maintenance support [12], many of which may exhibit the types of constraints described in the DTSPMS. Because of these side constraints, the DTSPMS is an appealing application to evaluate and compare various approaches and methodologies. *Indeed, industrial vehicle routing problems are rarely pure and often feature complex side-constraints, which make it increasingly important to compare existing algorithms on such complex applications.*

This paper proposes a Large Neighborhood Search (LNS) algorithm for the DTSPMS, since the constraint-based nature of LNS makes it a natural candidate to gracefully accommodate the problem-specific structure of the DTSPMS. Experimental results on difficult DTSPMS problems demonstrate the effectiveness of the algorithm. On the benchmarks provided by [13], our LNS algorithm matches or improves the best-known solutions in 65% of the instances, improving the best-known solution by more than 5% in one case. Moreover, the LNS algorithm is always within 2% of the best-known solution, indicating that the LNS algorithm is also robust.

The rest of this paper is organized as follows. Section 2 specifies the DTSPMS and describes the notations. Section 3 gives an overview of the overall algorithm. Section 4 presents the experimental results. Section 5 discusses related work and Section 6 concludes the paper.

2 Problem Formulation

This section defines the double traveling salesman problem with multiple stacks (DTSPMS) and various concepts used in the paper.

Customers The problem is defined in terms of n customers who are represented by the numbers $0, \ldots, n-1$ and four depots represented by d_0, d_1, d_2, d_3. The set $\{0, 1, \ldots, n-1, d_0, d_1, d_2, d_3\}$ represents all the sites considered in the problem. We use *Customers* to represent the set of customers and *Sites* to represent the set of sites (The distinction between customers and sites simplifies the formalization

of the problem and of the algorithm). We use $Customers^p$ and $Customers^d$ to denote the pickup and delivery customers respectively. Given a pickup customer i, its delivery counterpart is denoted by $@i$.

Travel Costs The *travel cost* between sites i and j is denoted by c_{ij}. Travel costs satisfy the triangular inequality $c_{ij} + c_{jk} \geq c_{ik}$.

Tour A traveling salesman tour, or tour for short, starts from depot d_0, visits all of the pickup customers, travels to depots d_1 and d_2, visits all of the delivery customers, and then returns to depot d_3. In other words, a tour is a sequence $\langle d_0, v_1^p, \ldots, v_n^p, d_1, d_2, v_1^d, \ldots, v_n^d, d_3 \rangle$. The travel cost of a tour denoted by *t(r)*, is the cost of visiting all its customers, i.e.,

$$t(r) = c_{d_0 v_1^p} + c_{v_1^p v_2^p} + \ldots + c_{v_{n-1}^p v_n^p} + c_{v_n^p d_1} + c_{d_2 v_1^d} + c_{v_1^d v_2^d} + \ldots + c_{v_{n-1}^d v_n^d} + c_{v_n^d d_3}.$$

Observe that a tour assigns a unique successor and predecessor to every site (except for the initial and final depots). The successor and predecessor of a site i in tour σ are denoted by $succ(i, \sigma)$ and $pred(i, \sigma)$. For simplicity, our definitions often assume an underlying tour σ and we use i^+ and i^- to denote the successor and predecessor of i in σ.

Departure Times The *departure time* of site i, denoted by δ_i, is defined recursively as

$$\begin{cases} \delta_{d_0} = 0 \\ \delta_i \ = \delta_{i^-} + c_{i^- i} \quad (i \in Sites \setminus d_0). \end{cases}$$

Pickup and Deliveries The pickup and deliveries are represented by a *precedence* constraint. The precedence constraint of $c \in Customers^p$ is satisfied if $\delta_c \leq \delta_{@c}$.

Stack Coupling The vehicle (or salesman) has available to it M stacks in which to store pickups. Each customer $c \in Customers$ is assigned a stack s_c and the stack coupling constraint is satisfied when $s_c = s_{@c}$ for all $c \in Customers$.

Last-in First-out The last-in first-out (LIFO) constraint requires that a delivery can occur if it is the most recent undelivered pickup on a stack. To specify the constraint formally, we need a number of operations on the set of stacks M: *push(c,s,M)* pushes customer c on stack s of M and returns the stack set, *top(s,M)* returns the customer on top of stack s of M, and *pop(s,M)* pops customer s of m and return the stack set. Given these operation, we can define a recursive algorithm to verify the LIFO constraint on a tour T (which is a sequence of sites). In the algorithm, we use :: to denote the concatenation of a site and a sequence of sites.

$$\begin{array}{ll} \text{LIFO}(\langle\rangle, M) & = true; \\ \text{LIFO}(d :: T, M) & = \text{LIFO}(T, M) \qquad\qquad (d \in \{d_0, d_1, d_2, d_3\}); \end{array}$$

$$
\begin{array}{lll}
\text{LIFO}(c :: T, M) & = \text{LIFO}(T, push(c, s_c, M)) & (c \in Customers^p); \\
\text{LIFO}(@c :: T, M) & = \text{if } \text{top}(s_{@c}, M) = c & (c \in Customers^d) \\
& \quad \text{then LIFO}(T, pop(s_{@c}, M)) & \\
& \quad \text{else } false. &
\end{array}
$$

The first rule states that the constraint is satisfied for an empty sequence. The second rule deals with the depots which do not affect the stacks. The third rule concerns the pickups which are pushed onto their chosen stack. The last rule deals with the deliveries. For a delivery $@c$, it must be the case that the customer on top of stack $s_{@c}$ is its pickup counterpart c.

Double Constraint Finally, there is the double tour constraint of the DTSPMS which states all pickups must occur prior to visiting d_1 and all deliveries must occur after visiting d_2. This constraint is satisfied for a customer $c \in Customers^p$ if $\delta_c < \delta_{d_1}$ and is satisfied for $c \in Customers^d$ if $\delta_c > \delta_{d_2}$.

The DTSPMS A solution to the DTSPMS is a tour T satisfying the LIFO constraints, double constraints, stack coupling, and pickup and delivery constraints, i.e.,

$$
\begin{cases}
s_i = s_{@i} & (i \in Customers) \\
\delta_i < \delta_{d_1} & (i \in Customers^p) \\
\delta_i > \delta_{d_2} & (i \in Customers^d) \\
\delta_i \leq \delta_{@i} & (i \in Customers^p) \\
\text{LIFO}(T, M) &
\end{cases}
$$

The DTSPMS problem consists of finding a solution σ which minimizes the the total travel cost, i.e., a solution σ minimizing the objective function specified by $f(\sigma) = t(\sigma)$.

3 The LNS Algorithm

Our algorithm is motivated by the success of large neighborhood search (LNS) on a variety of vehicle routing problems. LNS was proposed in [15] for the VRPTW, where it was shown particularly effective on the class 1 problems from the Solomon benchmarks, producing several improvements over the then best published solutions. It was later used as part of a two-stage approach for the VRPTW [1] and PDPTW [2]. The rest of this section describes the LNS algorithm in detail. In general, the algorithm adapts the heuristics and strategies described in [15], although it departs on a number of issues which are critical in generalizing LNS to the DTSPMS. The algorithm is presented incrementally, adding functionalities to remedy limitations observed in experiments.

3.1 The Neighborhood and the Evaluation Function

Given a solution σ, the neighborhood of LNS, denoted by $\mathcal{N}_R(\sigma)$, is the set of solutions that can be reached from σ by relocating the position and reassigning

the stacks of at most p pairs of customers (where p is a parameter of the implementation). Since LNS also uses subneighborhoods and explores them in a specific order, we use additional notations. In particular, $\mathcal{N}_R(\sigma, S)$ denotes the set of solutions that can be reached from σ by relocating and reassigning the customers in S. Also, given a partial solution σ with customers $Customers \setminus S$, $\mathcal{N}_I(\sigma, S)$ denotes the solutions that can be obtained by inserting the customers S in σ and assigning customers S to stacks in σ.

3.2 The Algorithm

At a high level, the LNS algorithm can be seen as a local search where each iteration selects a neighbor σ_c in $\mathcal{N}_R(\sigma_b)$ and accepts the move if $f(\sigma_c) < f(\sigma_b)$. It can be formalized as follows:

```
for(i ← 1...maxIterations) {
    SELECT σ_c ∈ N_R(σ_b);
    if f(σ_c) < f(σ_b) then
        σ_b ← σ_c;
}
```

In practice, it is important to refine and extend the above algorithm in three ways. The first modification consists of exploring the neighborhood by increasing number of allowed relocations. The second change generalizes the algorithm to a sequence of local searches. The third modification consists of exploring the sub-neighborhood $\mathcal{N}_R(\sigma_b, S)$ more exhaustively to find its best solution.

The overall algorithm is depicted in Figure 1. It performs a number of searches (line 1). Each search iterates over an increasing number of customers (line 2) and tries to improve the best found solution for a number of iterations (line 3). Each iteration consists in removing a number $2k$ customers, the algorithm always removing pickup and delivery pairs (line 4), and in selecting a best neighbor in $\mathcal{N}_R(\sigma_b, S)$ (line 5). Whenever a new best solution is found, the number of allowed iterations is re-initialized (line 8). In a sense, the algorithm is now very close to variable neighborhood search [9], but the neighborhood is hard to explore. It remains to describe how to select customers and how to implement line 5 in the above algorithm.

3.3 Selecting Customers to Relocate

The LNS algorithm adopts the customer selection strategy of [2] but modifies it to select pickup and delivery pairs. The implementation is depicted in Figure 2. It first selects a customer pair randomly (lines 1-2) and iterates lines 4-7 to remove the $k - 1$ remaining customer pairs. Each such iteration selects a pickup customer from S (the already selected customers) and ranks the remaining pickup customers according to a relatedness criterion (lines 4-5). The new customer to insert is randomly selected in line 6 and, once again, the algorithm

```
MINIMIZE(σ_o)
1  for l ← 1...maxSearches
2  do for k ← 1...p
3     do for i ← 1...maxIterations
4        do S ← SELECTCUSTOMERS(σ_o, k);
5           SELECT σ_n ∈ N_R(σ_o, S) SUCH THAT  f(σ_n) = min_{σ∈N_R(σ_o,S)} f(σ);
6           if f(σ_n) < f(σ_o)
7             then σ_o ← σ_n;
8                  i ← 1;
```

Fig. 1. The Abstract LNS Algorithm

```
SELECTCUSTOMERS(σ, k)
1  c ← {RANDOM(Customers^p)};
2  S ← {c, @c};
3  for i ← 2...k
4  do c ← RANDOM(S ∩ Customers^p);
5     ⟨c_0, ..., c_{|S|-i}⟩ ← Customers^p \ S SUCH THAT relateness(c, c_i) ≥ relateness(c, c_j)(i ≤ j);
6     r ← ⌊RANDOM([0,1])^β × |Customers^p \ S|⌋;
7     S ← S ∪ {c_r, @c_r};
```

Fig. 2. Selecting Customers in the LNS Algorithm

biases the selection toward related neighbors. The relatedness measure is simply defined as the distance between the customers, i.e.,

$$relateness(i, j) = c_{ij}.$$

3.4 The Exploration Algorithm

Our LNS algorithm uses a branch and bound algorithm to explore the selected sub-neighborhood. The algorithm is depicted in Figure 3. If the set of customers to insert is empty (line 1), the algorithm checks whether the current solution improves the best solution found so far (lines 2–3). Otherwise, it selects the customer pair whose best insertion degrades the objective function the most (line 4). The algorithm then computes all the partial solutions obtained by inserting c and $@c$ by increasing order of their travel costs (line 6). It then explores the resulting partial solutions recursively (lines 7–9). Also, observe that only the partial solutions whose lower bounds are better than the best solution found so far are explored by the algorithm (line 8). The lower bound satisfies the inequality $\text{BOUND}(\sigma, S) \leq \min_{\sigma' \in \mathcal{N}_I(\sigma,S)} f(\sigma')$.

The bounding function is the cost of a minimum spanning 1-tree bound of the traveling salesman problem. The insertion graph vertices are the customers. Given a solution σ over customers $C = \cup_{r \in \sigma} cust(r)$ and a set S of vertices to insert, the insertion graph edges come from three different sets:

```
EXPLORE(σ_c, S, σ_b)
1  if S = ∅
2    then if f(σ_c) < f(σ_b)
3           then σ_b ← σ_c;
4    else  c ← arg-max_{c∈S} min_{σ∈N_I(σ,{c,@c})} f(σ);
5          S_c ← S \ {c, @c};
6          ⟨σ_0, ..., σ_k⟩ ← N_I(σ, {c, @c})WHERE f(σ_i) ≤ f(σ_j) (i ≤ j);
7          for i ← 1...k
8          do if BOUND(σ_i, S_c) < f(σ_b)
9               then EXPLORE(σ_i, S_c, σ_b);
```

Fig. 3. The Branch and Bound Algorithm for Exploring the Neighborhood.

```
DISEXPLORE(σ_c, S, σ_b, d, dmax)
 1  if d ≤ dmax
 2    then if S = ∅
 3           then if f(σ_c) < f(σ_b)
 4                  then σ_b ← σ_c;
 5    else  c ← arg-max_{c∈S} min_{σ∈N_I(σ,{c,@c})} f(σ);
 6          S_c ← S \ {c, @c};
 7          ⟨σ_0, ..., σ_k⟩ ← N_I(σ, {c, @c})WHERE f(σ_i) ≤ f(σ_j) (i ≤ j);
 8          for i ← 1...k
 9          do if BOUND(σ_i, S_c) < f(σ_b)
10               then DISEXPLORE(σ_i, S_c, σ_b, d, dmax);
11                    d ← d + 1;
```

Fig. 4. The Branch and Bound Algorithm with a Discrepancy Limit.

1. the edges already in σ;
2. all the edges between customers in S;
3. all the feasible edges connecting a site from $Sites \setminus S$ and a customer from S.

3.5 Using Discrepancy Search

For many problems, finding the best reinsertion is too time-consuming. Our algorithm (depicted in Figure 4 uses discrepancy search (e.g., [10]) to explore only a small part of the search tree. More precisely, the algorithm allows up to $dmax$ discrepancies. The changes are in line 1, which tests whether the discrepancy limit is reached and in line 11, which increases the number of discrepancy. Note that the tree is not binary and the heuristic selects the insertion points by increasing lower bounds. Observe also that the neighborhood $\mathcal{N}_I(\sigma, \{c, @c\})$ is of size $O(N^2M)$, of which only a portion is explored by LDS.

```
DisExplore(σ_c, S, σ_b, d, dmax)
 1  if d ≤ dmax
 2    then if S = ∅
 3           then if f(σ_c) < f(σ_b)
 4                  then σ_b ← σ_c;
 5    else  c ← arg-max_{c∈S} min_{σ∈N_I(σ,{c,@c})} f(σ);
 6          S_c ← S \ {c, @c};
 7          for m ∈ M
 8          do s_c ← m;
 9             s_@c ← m;
10             ⟨σ_0, ..., σ_k⟩ ← N_I(σ, {c, @c}) WHERE f(σ_i) ≤ f(σ_j)  (i ≤ j);
11             for i ← 1...k
12             do if Bound(σ_i, S_c) < f(σ_b)
13                  then DisExplore(σ_i, S_c, σ_b, d, dmax);
14                       d ← d + 1;
```

Fig. 5. Limited Discrepancy Strategy with Explicit Stack Assignments.

3.6 Decoupling Decision Variables

The algorithm depicted in Figure 4 was not particularly effective. An analysis of its behavior revealed that the stack assignments were deteriorating the search. Indeed, there were many moves that differed only in the stack assignment, had the same objective value, and incremented the discrepancy count, so only a small portion of the search space was explored. The reason, of course, comes from the fact that stack allocations have no *direct* effect on the objective function: they simply restrict the set of subsequent moves. To address the bias on the discrepancy count coming from stack allocation, the new version of algorithm (see Figure 5) iterates over all stack allocations which are not longer part of the discrepancy computation. In other words, the algorithm makes decisions in two steps: First, it chooses the customer pair (line 5) and then iterates over the stack allocation (lines 7–9). Then, the algorithm generates the insertion points (line 10) and explores them recursively (lines 12–14). Note that the discrepancies only concern the insertion points, not the stack allocations.

3.7 Restarts

The algorithm presented in Figure 5 produced high-quality results on many instances but the experimental results reported a high variance on almost all the benchmarks. These results motivated a careful study of the execution of large neighborhood search on these problems. The analysis yielded the observation that the search tended to discover its local minimum very early in the search process. Then, it spent considerable computation time trapped in one section of the search space and was unable to progress towards better solutions. This behavior was not observed in LNS algorithms for other routing problems and is probably due to the stack constraints which reduce the flexibility of LNS

```
MINIMIZE()
 1  for l ← 1...maxStarts
 2  do σ_o ← INITSOLUTION();
 3     for l ← 1...maxSearches
 4     do for k ← 1...p
 5        do for i ← 1...maxIterations
 6           do S ← SELECTCUSTOMERS(σ_o, k);
 7              SELECT σ_n ∈ N_R(σ_o, S) SUCH THAT  f(σ_n) = min_{σ∈N_R(σ_o,S)} f(σ);
 8              if f(σ_n) < f(σ_o)
 9                then σ_o ← σ_n;
10                     i ← 1;
11  return σ_o;
```

Fig. 6. The Abstract LNS Algorithm Revisited

considerably. This behavior was addressed by introducing a multistart strategy (see Figure 6): the search is restarted from an initial random solution periodically. The random initial solution is obtained by using a single stack.

3.8 Constraint Propagation

It also interesting to discuss how the constraints are used in the LNS algorithm. The primary role of constraints in the LNS algorithm is to prune the set of feasible insertions and stack assignments. More precisely, every time a customer is assigned a stack or a position on the tour, the constraints are invoked individually to prune the possible assignments and tour insertions of the remaining customers. The constraints take advantage of the search structure (i.e., the fact that (pickup,delivery) pairs are inserted sequentially) to implement efficient pruning algorithms. Each insertion of a customer generates a new insertion point for each customer not on the tour. This insertion point must check for feasibility in order to be added as a possible insertion for each customer. The presentation below explains the behavior of the constraints for the final algorithm, although they can be used in other ways.

Pickup and Delivery This constraint ignores stack assignments, since it does not interact with stack assignments. Without loss of generality (just reverse the logic), assume the first customer inserted of a pair is the pickup customer c. Upon this insertion, the constraint eliminates all insertion points for @c prior to this insertion point on the tour in $O(n)$ time. Interestingly, in the DTSPMS, this operation turns out to take $O(1)$ time because the double tour constraint.

Stack Coupling Whenever a customer pair is inserted onto a tour, its location can reduce the set of feasible stack assignments. However, since the pairs are inserted onto tours *after* the stacks are assigned in the LNS algorithm presented here, this constraint does nothing when a customer insertion occurs. When one

customer c of a pair is assigned a stack m, this constraint removes all assignments but m from $@c$ in $O(m)$ time.

Double Constraint This constraint is only invoked on the initial generation of feasible insertion points for LNS: It is always satisfied after that.

LIFO Constraint This constraint is the most complicated of the four. First, each customer c uses M variables, denoted by $top(c, m)$, to maintain the LIFO constraint. The variable $top(c, m)$ represents the customer at the top of the m-th stack after customer c is visited. This allows us to use only $O(M)$ space per customer as opposed to (Mn) if the algorithm would store the entire stack. It makes it possible to improve the computational efficiency as the algorithm only needs to update $top(c, m)$ for those customers and stacks that are changed by a stack assignment or an insertion.

As stack assignments occur prior to tour assignments, the LIFO constraint does nothing when a customer is assigned a stack in this LNS implementation. It is during the pair insertion of customers that this constraint takes advantage of the fact that pairs are inserted sequentially. Under this scenario, when the first customer c of a pair is inserted, the feasible insertions of $@c$ are updated. After $@c$ is inserted, the appropriate *top* values are updated. Let us describe these operations in more details assuming, once gain without loss of generality, that c is the pickup customer.

After c is inserted, the only insertion points to update are those of $@c$. Clearly, the insertion point between c and c^+ (the successor of c) remains feasible as do all subsequent insertion points whose customer is on a stack other than s_c. Upon reaching an insertion point whose customer is on s_c, the algorithm needs to perform a case analysis. If this customer is a delivery customer, then every subsequent insertion point for $@c$ is infeasible by definition. If this customer, say p, is a pickup customer, then every subsequent insertion point up to $@p$ is infeasible. After which all subsequent insertion points for $@c$ remain feasible until an insertion point whose customer is on s_c. At which point the original case analysis is epeated until the last insertion point is checked or a delivery customer is observed. This operation takes $O(n)$ time. This is not an issue, since the update of *top* (described next) also takes $O(n)$ time,

After $@c$ is inserted, *top* needs to be updated. First, $top(c, s_c) = c$ and $top(c, m) = top(c^-, m)$ for $m \in M \setminus s_c$, where c^- denotes the predecessor of c. Second, $top(@c, s_c) = top(c^-, s_c)$ and $top(@c, m) = top(@c^-, m)$ for $m \in M \setminus s_c$. Finally, the remaining *top* entries are updated by calling $\text{UPDATETOP}(c, c^+, s_c)$ defined as follows:

$\text{UPDATETOP}(p, u, m)$
1 **while** $s_u \neq m$
2 **do** $top(u, s_u) \leftarrow p$;
3 $u = u^+$;
4 **if** $u \in Customers^p$
5 **then** $\text{UPDATETOP}(p, @u, m)$;

This algorithm takes as arguments a pickup customer to update top (p), a customer to start updating top (u) and a stack to update (m). It is designed to propagate the value of $top(p, m)$ forward to all places that need to be updated (i.e. propagating c as a top value all the way to @c). It updates all the *top* values of m to p for subsequent customers until a customer, u, with the same stack is found (lines 1-3). If u is a pickup, the algorithm skips ahead to @u, which is where the updating of *top* needs to restart (line 5). If u is a delivery, by definition it must be @c and all top values have been updated appropriately. The nice aspect of this operation is that it only views those values of *top* that actually need to be updated, so it is fully incremental. As the actual stack is not stored at every customer, we can skip those top variables where c is not on the top of the stack.

4 Experimental Results

This section presents our experimental results on the only set of publicly available benchmarks for the Double TSP with Multiple Stacks that were provided by [13]. All results were obtained on an Intel 2.4Ghz chip using Java version 1.6 and double precision numbers. For all the experimental results reported, the following parameters for Large Neighborhood Search were used: $p = 15$, $maxIterations = 100$, $dmax = 5$, and $\beta = 15$. The algorithms is restarted every 90 seconds when restarts are used. Each problem was solved 10 times for a maximum of fifteen minutes.

4.1 The Benchmarks

As mentioned, the benchmarks were taken from [13]. They are created by generating two sets of thirty-three customers in a 100x100 square. The pickup and delivery pairs were randomly assigned between the two sets. All four depots are located at coordinate (50,50) and all travel distances are produced using the Euclidean distance between the sites. These problems include three stacks.

4.2 Solution Quality Without Restarts

Table 4.2 provides results when no restarts are used. As seen from Table 1, the algorithm finds 10 new best solutions over [13] within the same time limits (15-20 minutes in [13]). However, one can easily see the high variance in the the worst and average results. The worst results across the runs can be as bad as almost 15% of the best known solution, while the average results can be as much as 8.2% away. These are precisely those observation that motivated us to include a restart strategy from a random initial solution.

4.3 Solution Quality With Restarts

Table 4.3 provides results when LNS is restarted every ninety seconds with a random initial solution utilizing a single stack. The algorithm improves the best-known solution on 12 instances and matches another one. Moreover, the best

Problem	Best	%	Worst	%	Average	%	[13] Result
R00	1098.48	2.7%	1199.77	12.2%	1150.85	7.6%	1069
R01	**1064.77**	-0.7%	1154.22	7.6%	1112.08	3.7%	1072
R02	1076.37	0.5%	1147.51	7.2%	1116.19	4.3%	1070
R03	1113.16	0.2%	1240.99	11.6%	1182.79	6.4%	1111
R04	1096.76	0.5%	1194.02	9.5%	1150.29	5.5%	1090
R05	**1029.75**	-2.5%	1144.82	8.4%	1064.37	0.9%	1055
R06	**1104.73**	-1.3%	1281.42	14.6%	1186.45	6.1%	1118
R07	1125.77	0.6%	1268.25	13.4%	1200.85	7.3%	1118
R08	1113.09	0.2%	1258.66	13.2%	1202.35	8.2%	1111
R09	**1096.38**	-0.9%	1227.32	10.9%	1154.40	4.4%	1106
R10	**1016.04**	-0.5%	1170.81	14.6%	1083.66	6.1%	1021
R11	**1034.72**	-0.6%	1179.35	13.4%	1083.70	4.1%	1040
R12	1135.42	2.0%	1250.57	12.3%	1196.40	7.5%	1113
R13	**1090.87**	-1.0%	1199.07	8.8%	1136.79	3.1%	1102
R14	**1050.30**	-0.9%	1099.60	3.4%	1075.53	1.5%	1059
R15	1177.64	1.3%	1274.86	9.6%	1225.28	5.4%	1162
R16	1115.45	0.9%	1204.34	9.0%	1158.88	4.8%	1105
R17	1113.96	1.6%	1238.58	13.0%	1167.56	6.5%	1096
R18	**1161.68**	-1.6%	1284.58	8.8%	1235.91	4.7%	1180
R19	**1077.11**	-41.%	1230.21	9.5%	1155.19	2.8%	1123

Table 1. Solution Quality of LNS for the DTSPMS: Results without Restarts.

solution of the algorithm is never worse than 2% of the best-known solution and almost always well below 1%. It can also be seen from these results that the variance has been reduced considerably. On three instances, the average solution quality is lower than the best-known solution and the average solution is never worse than 4.6% of the best-known solution. LNS is thus a particularly effective solution technique for the DTSPMS.

5 Discussion and Related Work

Recent years has seen an increase in attention to vehicle routing and traveling salesman problems variants where the way in which commodities are loaded onto a vehicle has an influence on the tours the vehicle is allowed to traverse. Most of the literature (except [13]) has considered variations that do not include the Double TSP constraint that is described here. [4] considers the pickup and delivery traveling salesman problem with LIFO constraints on a single stack where a delivery can occur at any time as long as the LIFO order is preserved. One of the major contributions here is a number of customized traveling salesman neighborhood move operations for this problem as many of the traditional TSP move operations do not preserve feasibility. It tests the effectiveness of the operations by embedding them in a variable neighborhood search. Reference [5] considers the same class of problems as [4] but instead focuses on global optimization

Problem	Best	%	Worst	%	Average	%	[13] Result
R00	**1069.12**	0.0%	1142.69	6.8%	1104.74	3.3%	1069
R01	**1043.64**	-2.7%	1108.02	3.4%	1076.13	0.4%	1072
R02	1074.29	0.4%	1118.25	4.5%	1095.39	2.3%	1070
R03	1129.29	1.6%	1197.48	7.7%	1155.80	4.0%	1111
R04	**1079.93**	-1.0%	1133.60	3.9%	1101.66	1.0%	1090
R05	**1001.01**	-5.2%	1066.34	1.0%	**1038.04**	-1.6%	1055
R06	1120.50	0.2%	1236.05	10.6%	1169.29	4.6%	1118
R07	**1114.55**	-0.4%	1218.62	8.9%	1166.66	4.3%	1118
R08	1133.43	2.0%	1173.51	5.6%	1154.04	3.9%	1111
R09	**1093.70**	-1.2%	1172.62	6.0%	1123.95	1.5%	1106
R10	**1016.04**	-0.5%	1135.15	11.2%	1059.88	3.7%	1021
R11	1046.35	0.6%	1090.07	4.8%	1062.12	2.1%	1040
R12	1115.30	0.2%	1200.79	7.8%	1154.77	3.7%	1113
R13	**1082.01**	-1.8%	1142.94	3.6%	1115.40	1.2%	1102
R14	**1032.83**	-2.5%	1099.57	3.4%	1067.35	0.8%	1059
R15	1171.75	0.8%	1215.14	4.6%	1195.80	2.8%	1162
R16	**1087.46**	-1.6%	1169.50	5.8%	1133.27	2.5%	1105
R17	**1077.25**	-1.7%	1139.69	3.9%	1115.09	1.7%	1096
R18	**1153.79**	-2.3%	1201.51	1.8%	**1172.92**	-0.7%	1180
R19	**1097.66**	-2.3%	1135.78	1.1%	**1111.97**	-1.1%	1123

Table 2. Solution Quality of LNS for the DTSPMS: Results with Restarts.

approaches. They describe a branch-and-cut algorithm that is able to push the size of tractably solvable problems from 25 to 36 customers by introducing a novel set of cuts. [16] also considers global search optimization techniques for this vehicle routing problems with LIFO constraints on a single pickup and delivery stack. This paper is motivated by real problems, so they include a much richer set of constraints than is typically considered in the literature and they describe techniques for generating realistic benchmarks. They propose decomposing the problem into component vehicles using a column generation approach and heuristics to find good solutions to routing the customers assigned to the individual vehicles.

Reference [7] considers a vehicle routing problem with an embedded three-dimensional bin packing problem. The bin packing problem implies the existence of a LIFO constraint as a delivery can be made only if it is reachable without moving any other packages in the vehicle. The paper approaches the problem by separating the bin packing problem from the routing problem, i.e. their tabu search first finds a bin packing arrangement and then searches for feasible routing solutions that obey the bin packing. They found that allowing the tabu search to explore infeasible regions of the search space improved the quality of their results. [11] and [8] consider similar problems as [7] except that the bin packing problem is two-dimensional. They consider branch-and-cut algorithms and tabu search respectively. It is important to note that the multi-stack traveling sales-

man problem considered here may be thought of as a three or two dimensional bin packing problem with the special property that all items to be packed are of the same size.

The most related reference to this paper is [13] on which the description of a single vehicle, multi-stack with a Double TSP constraint is first provided. The reference evaluates three different local search heuristics (tabu search, simulated annealing, and steepest descent) with a variety of different of parameter settings and suggests that simulated annealing was their most effective approach.

Finally, there is recent work on similar problems where the packing order is constrained by first-in first-out considerations (FIFO) constraints (i.e., a queue) in situations such as dial-a-ride where early customers are not satisfied if later pickups are delivered earlier than they are. Reference [6] suggests a number of local search neighborhoods for handling the FIFO constraint in the context of tabu search.

6 Conclusion

This paper proposed a large neighborhood search algorithm for the Double Traveling Salesman Problem with Multiple Stacks (DTSPMS). The algorithm minimizes the total travel distance given pickup and delivery, stack coupling, LIFO, and double tour constraints. Experimental results show the effectiveness of the approach which produced many new best solutions on benchmarks in the literature. The paper also demonstrates positively that LNS is easily modified to handle side assignments such as stack packing order.

There are many open issues that deserve attention. As research moves to large-scale problems involving several hundreds or thousands of customers, scaling the algorithms raise new interesting challenges that were not systematically studied here. One promising area is in decomposition techniques such as in [3] where new techniques for decomposing constraints of the DTSPMS in a natural way will need to be developed. It would be interesting to study the impact of various parameters on the behavior of the algorithm and to study how to tune these decisions dynamically during search. Of course, it will be interesting to study LNS for DTSPMS on more complex vehicle routing problems including multiple vehicles, time windows, and capacities and considering how the number of stacks impacts the difficulty of the problem sets. Finally, it will be interesting to study other side constraints such as first-in first-out (FIFO) constraints and side decision variables such as multiple pickup and delivery problems where there are decision variables associated with assigning pickup points for each delivery.

Acknowledgments This work was supported in part by the U.S. Department of Homeland Security's National Infrastructure Analysis and Simulation Center (NISAC) Program, by NSF award DMI-0600384, and ONR Award N000140610607.

References

1. Russell Bent and Pascal Van Hentenryck. A Two-Stage Hybrid Local Search for the Vehicle Routing Problem with Time Windows. *Transportation Science*, 38(4):515–530, 2004.
2. Russell Bent and Pascal Van Hentenryck. A Two-Stage Hybrid Algorithm for Pickup and Delivery Vehicle Routing Problems with Time Windows. *Computers and Operations Research*, 33 (4):875–893, 2006.
3. Russell Bent and Pascal Van Hentenryck. Randomized Adaptive Spatial Decoupling For Large-Scale Vehicle Routing with Time Windows. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI)*, Vancouver, Canada, 2007.
4. Francesco Carrabs, Jean-Francois Cordeau, and Gilbert Laporte. Variable neighborhood search for the pickup and delivery traveling salesman with lifo loading. *INFOR Journal on Computing*, 19:618–632, 2007.
5. Jean-Francois Cordeau, Manuel Iori, Gilbert Laporte, and Juan Gonzalez. A branch and cut algorithm for the pickup and delivery traveling salesman problem with lifo loading. *Networks*, to appear.
6. Gunes Erdogan, Jean-Francois Cordeau, and Gilbert Laporte. The pickup and delivery traveling salesman problem with first-in-first-out loading. *CORS Optimization Days*, 2008.
7. Michel Gendreau, Manuel Iori, Gilbert Laporte, and Silvano Martello. A tabu search algorithm for a routing and container loading problem. *Transportation Science*, 40:342–350, 2006.
8. Michel Gendreau, Manuel Iori, Gilbert Laporte, and Silvaro Martello. A tabu search heuristic for the vehicle routing problem with two-dimensional loading constraints. *Networks*, 51 (1):4–18, 2007.
9. Pierre Hansen and Nenad Mladenovic. An Introduction to Variable Neighborhood Search. In *Meta-heuristics, Advances and Trends in Local Search Paradigms for Optimization*, pages 433–458, Boston, Massachusetts, 1998. Kluwer Academic Publishers.
10. William Harvey and Matthew Ginsberg. Limited Discrepancy Search. In *Proceedings of the Forteenth International Joint Conference on Artifical Intelligence (IJCAI)*, pages 607–615, Montreal, Canada, 1995.
11. Manuel Iori, Juan Jose Salazar Gonzalez, and Daniele Vigo. An exact approach for the vehicle routing problem with two-dimensional loading constraints. *Transportation Science*, 41:253–264, 2007.
12. William Nanry and J. Wesley Barnes. Solving the Pickup and Delivery Problem with Time Windows Using Reactive Tabu Search. *Transportation Research Part B*, 34:107–121, 2000.
13. Hanne Peterson. Heuristic solution approaches to the double tsp with multiple stacks. *CORS Optimization Days*, 2006.
14. Martin Savelsbergh and Marc Sol. The General Pickup and Delivery Problem. *Transportation Science*, 29:107–121, 1995.
15. Paul Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In *Proceedings of the Fourth International Conference on the Principles and Practice of Constraint Programming (CP)*, pages 417–431, Pisa, Italy, 1998.
16. Hang Xu, Zhi-Long Chen, Srinivas Rajagopal, and Sundar Arunapuram. Solving a practical pickup and delivery problem. *Transportation Science*, 37:347–364, 2003.