CIRRELT

Centre interuniversitaire de recherche
sur les réseaux d'entreprise, la logistique et le transport

Interuniversity Research Centre
on Enterprise Networks, Logistics and Transportation

Large Neighborhood Search for the Single Vehicle Pickup and Delivery Problem with Multiple Loading Stacks

Jean-François Côté
Michel Gendreau
Jean-Yves Potvin

November 2009

CIRRELT-2009-47

Bureaux de Montréal :
Université de Montréal
C.P. 6128, succ. Centre-ville
Montréal (Québec)
Canada H3C 3J7
Téléphone : 514 343-7575
Télécopie : 514 343-7121

Bureaux de Québec :
Université Laval
2325, de la Terrasse, bureau 2642
Québec (Québec)
Canada G1V 0A6
Téléphone : 418 656-2073
Télécopie : 418 656-2624

www.cirrelt.ca

Large Neighborhood Search for the Single Vehicle Pickup and Delivery Problem with Multiple Loading Stacks

Jean-François Côté[1,2], Michel Gendreau[1,3], Jean-Yves Potvin[1,2,*]

[1] Interuniversity Research Centre on Enterprise Networks, Logistics and Transportation (CIRRELT)

[2] Department of Computer Science and Operations Research, Université de Montréal, P.O. Box 6128, Station Centre-ville, Montréal, Canada H3C 3J7

[3] Department of Mathematics and Industrial Engineering, École Polytechnique de Montréal, P.O. Box 6079, Station Centre-ville, Montréal, Canada H3C 3A7

Abstract. This paper studies a single vehicle pickup and delivery problem with loading constraints. In this problem, the vehicle contains a number of (horizontal) stacks of finite capacity for loading items from the rear of the vehicle. Each stack must satisfy a last-in-first-out constraint where any new item must be loaded on top of a stack and any unloaded item must be on top of its stack. A large neighborhood search is proposed for solving this problem. Computational results are reported on different types of randomly generated instances. Results are also reported on benchmark instances for two special cases of our problem and a comparison is provided with state-of-the-art methods.

Keywords. Vehicle routing, pickup, delivery, loading, multiple stacks, large neighborhood search.

Acknowledgements. Financial support for this work was provided by the Natural Sciences and Engineering Council of Canada (NSERC). This support is gratefully acknowledged.

---

* Corresponding author: Jean-Yves.Potvin@cirrelt.ca

Dépôt légal – Bibliothèque et Archives nationales du Québec,
Bibliothèque et Archives Canada, 2009

1 Introduction

In this work, we consider a single vehicle Pickup and Delivery Problem (PDP), where items are loaded in (horizontal) stacks from the rear of the vehicle. Each customer request has a pickup location, where the items are loaded, and a delivery location, where the items are unloaded. The loading and unloading operations in each stack must satisfy a Last-In-First-Out (LIFO) constraint. That is, any new item must be loaded on top of a stack and an item can be unloaded only if it is on top of its stack. This constraint prevents reordering of the items along the route, which is desirable when large items are transported. It is also assumed that the stacks are independent (i.e., each item fits within a single stack) and are of finite capacity. Furthermore, the demand at each customer cannot be split among different stacks. The goal is to design a least-cost route for the vehicle, starting from and ending at a central depot, that serves all customer requests while satisfying the side constraints, namely, the precedence constraint between the pickup and delivery location of each request, the capacity constraint of each stack and the LIFO loading constraint. This problem will be referred to as the single vehicle Pickup and Delivery Problem with Multiple Stacks (1-PDPMS) in the following.

This problem is a generalization of the Traveling Salesman Problem with Pickup and Delivery and LIFO Loading constraint (TSPPDL), where the vehicle contains a single stack of infinite capacity [2, 3]. It is also a generalization of the Double Traveling Salesman Problem with Multiple Stacks (DTSPMS), where all pickups, and then all deliveries, are performed in two different routes, and where each stack must satisfy the LIFO loading constraint [12]. Problems that generalize ours are the Multi-Pile Vehicle Routing Problem (MPVRP) [5] where items can extend over a number of stacks, and various two- and three-dimensional PDPs where the items have different shapes that must be loaded within a finite surface or volume [6, 7, 8, 9, 10].

A Large Neighborhood Search (LNS) [1, 16] is proposed here to solve the 1-PDPMS. In this iterative method, a large neighborhood of the current solution is obtained by removing a number of customer requests and by reinserting them to obtain a new, hopefully better, solution. This approach, based on the ruin-and-recreate principle [15], involves a number of removal and reinsertion operators, including innovative ones, like a stack-based removal operator and an insertion operator based on the adaptation of a generalized regret measure [13] that accounts for multiple stacks.

It is empirically demonstrated that the adaptation of the LNS framework to our problem produces high quality solutions. Furthermore, it outperforms specialized state-of-the-art methods for particular cases of our problem, namely the TSPPDL and DTSPMS. The remainder of the paper is organized as follows. The problem is formally introduced in Section 2. Then, our problem-solving methodology is described in Section 3. Computational results are reported in Section 4. Finally, a conclusion follows.

2 Problem Formulation

The 1-PDPMS can be formally stated as follows. Let $G = (V, A)$ be a complete graph where $V = \{0, 1, ..., 2n\}$ is the vertex set and A is the arc set. Vertex 0 stands for the depot while vertices i and $n + i$ are the pickup and delivery locations of customer request i, $1 \leq i \leq n$. We denote $P = \{1, ..., n\}$ and $D = \{n+1, ..., 2n\}$ the set of pickup and delivery locations, respectively. With each pickup location $i \in P$ is associated a demand d_i. We assume that a demand $d_0 = 0$ is associated with the depot and a demand $-d_i$ with delivery location $n + i \in D$. We also have a cost c_{ij} on each arc $(i, j) \in A$.

The vehicle contains a set $M = \{1, 2, ..., m\}$ of loading stacks, each of capacity Q, to transport the demand between pickup and delivery locations. The goal is to find a least-cost route for the vehicle, starting from and ending at the depot, that serves all customer requests while satisfying the side constraints.

This problem can be mathematically formulated using the following decision variables:

- x_{ij} is 1 if vertex j is visited immediately after vertex i, 0 otherwise, $i, j \in V$, $i \neq j$;
- y_{ik} is 1 if the demand at pickup location i is loaded in stack k, 0 otherwise, $i \in P$, $k \in M$;
- $0 \leq u_i \leq 2n$ is the position of vertex i in the route, $i \in V$ (with $u_0 = 0$);
- $0 \leq s_{ik} \leq Q$ is the load of stack k upon leaving vertex i, $i \in V$, $k \in M$ (with $s_{0k} = 0$, $k \in M$).

We then have:

$$\min \sum_{i \in V} \sum_{\substack{j \in V \\ j \neq i}} c_{ij} x_{ij} \tag{1}$$

subject to

$$\sum_{j \in V} x_{ij} = 1, \ \forall i \in V \tag{2}$$

$$\sum_{j \in V} x_{ji} = 1, \ \forall i \in V \tag{3}$$

$$\sum_{k \in M} y_{ik} = 1, \ \forall i \in P \tag{4}$$

$$u_j \geq u_i + 1 - 2n(1 - x_{ij}), \ \forall i \in V, \ \forall j \in P \cup D \tag{5}$$

$$u_{n+i} \geq u_i + 1, \ \forall i \in P \tag{6}$$

$$s_{jk} \geq s_{ik} + d_j y_{jk} - Q(1 - x_{ij}), \ \forall i \in V, \ \forall j \in P, \ \forall k \in M \tag{7}$$

$$s_{(n+j)k} \geq s_{ik} + d_{n+j}y_{jk} - Q(1 - x_{i(n+j)}),\ \forall i \in V,\ \forall j \in P,\ \forall k \in M \quad (8)$$

$$s_{(n+j)k} \geq s_{jk} + d_{n+j}y_{jk} - Q(1 - y_{jk}),\ \forall j \in P,\ \forall k \in M \quad (9)$$

$$u_0 = 0 \quad (10)$$

$$1 \leq u_i \leq 2n,\ \forall i \in P \cup D \quad (11)$$

$$s_{0k} = 0,\ \forall k \in M \quad (12)$$

$$0 \leq s_{ik} \leq Q,\ \forall i \in P \cup D,\ \forall k \in M \quad (13)$$

In this formulation, the objective function (1) is aimed at minimizing the total cost which corresponds here to the distance traveled by the vehicles. Each vertex is visited exactly once through constraints (2) and (3). Constraint (4) states that the demand of each pickup location is loaded in exactly one stack. The position of each vertex in the route is defined through (5). The precedence constraint between the pickup and delivery locations is found in (6). Constraints (7) and (8) define the status of the stacks after each pickup and delivery. The LIFO loading constraint is stated in (9). Constraints (10) and (11) define the vertex positions along the route. Finally, constraints (12) and (13) take into account the capacity of each stack.

This model allows $m!$ different representations of the same solution (by interchanging the contents of the stacks). To break this symmetry, the two following constraints are added:

$$y_{11} = 1 \quad (14)$$

$$y_{ik} \leq \sum_{j=1}^{i-1} y_{j(k-1)} \text{ for } k > i\ , \forall i \in P \quad (15)$$

Constraint (14) forces the demand at pickup location 1 to be on stack 1. Then, constraint (15) states that the demand at pickup location i can be loaded in stack $k > i$ only if stack $k - 1$ is used.

3 Large Neighborhood Search

In the two following subsections, the removal and insertion operators of our LNS algorithm are described. Then, the iterative search mechanism based on these operators is presented.

3.1 Removal operators

These operators remove q customer requests from the current route. Clearly, a feasible route remains feasible after their application. These operators are described in the following subsections.

3.1.1 Customer-based operators

Random removal

This is a very straightforward operator where q requests are removed at random.

Distance-based removal

This operator is inspired from [16], where related requests are removed, based on different metrics. Our distance-based operator is described in the pseudo-code that follows, where S denotes the current solution and p(i) and d(i) return the pick-up and delivery vertex of request i, respectively.

1. $i \leftarrow$ RandomRequest(S) ;
2. $L \leftarrow \{i\}$;
3. $S \leftarrow S \setminus \{p(i), d(i)\}$;
4. While $\mid L \mid < q$ do
   - 4.1 $i \leftarrow$ Random(L) ;
   - 4.2 $B \leftarrow \emptyset$;
   - 4.3 For each request $j \in S$ do
     - 4.3.1 $b_j \leftarrow c_{p(i)p(j)} + c_{d(i)d(j)}$;
     - 4.3.2 $B \leftarrow B \cup \{j\}$;
   - 4.4 Sort B in increasing order of b_j ;
   - 4.5 $r \leftarrow$ RandomNumber(0, 1) ;
   - 4.6 $pos \leftarrow \lceil \mid B \mid \cdot r^d \rceil$;
   - 4.7 Select request j at position pos in B ;
   - 4.8 $L \leftarrow L \cup \{j\}$;
   - 4.9 $S \leftarrow S \setminus \{p(j), d(j)\}$;
5. Return L.

Starting with a randomly chosen request, which starts the whole procedure, the removal of the next requests is (probabilistically) biased toward those that are close to one of the previously removed requests, based on the distance metric. Parameter d in step 4.6 controls the intensity of the bias. Namely, a high value for parameter d strongly favors the removal of requests that are close to previously removed requests (and conversely). Based on preliminary experiments, this parameter was set to 6.

3.1.2 Route-based removal

The goal of this operator is to remove a sequence of consecutive vertices from the route. Clearly, if the vertex is a pickup then the corresponding delivery also needs

to be removed (and conversely). In the pseudo-code below, pred_req(i) returns the request whose pickup or delivery vertex is the immediate predecessor of the pickup vertex of request i (it returns 0 if this predecessor is the depot). Similarly, succ_req(i) returns the request whose pickup or delivery vertex is the immediate successor of the pickup vertex of request i.

1. $i \leftarrow$ RandomRequest(S) ;
2. $L \leftarrow \emptyset$;
3. While $\mid L \mid < q - 1$ do
   - 3.1 $j \leftarrow pred\_req(i)$;
   - 3.2 If $j \neq 0$ then
     - 3.2.1 $L \leftarrow L \cup \{j\}$;
     - 3.2.2 $S \leftarrow S \setminus \{p(j), d(j)\}$;
   - 3.3 if $\mid L \mid < q - 1$ then
     - 3.3.1 $j \leftarrow succ\_req(i)$;
     - 3.3.2 If $j \neq 0$ then
       $L \leftarrow L \cup \{j\}$;
       $S \leftarrow S \setminus \{p(j), d(j)\}$;
4. $L \leftarrow L \cup \{i\}$;
5. $S \leftarrow S \setminus \{p(i), d(i)\}$;
6. Return L.

A request i is first randomly chosen and $q-1$ requests around i are then removed by first considering the predecessor, and then, the successor of its pickup vertex. If the predecessor happens to be the depot, then the remaining requests are removed by considering only the successors (and conversely). At the end, request i is also removed.

3.1.3 Stack-based removal

Here, customer requests are removed from the current solution with the distance-based removal operator (see subsection 3.1.1), except that the distance metric is based on the difference between their positions in the stack. Thus, given two requests i and j in the same stack, step 4.2.1 becomes:

$$b_j \leftarrow \mid pos(i) - pos(j) \mid ;$$

where $pos(i)$ and $pos(j)$ are the positions of the pickup vertices of requests i and j in the stack.

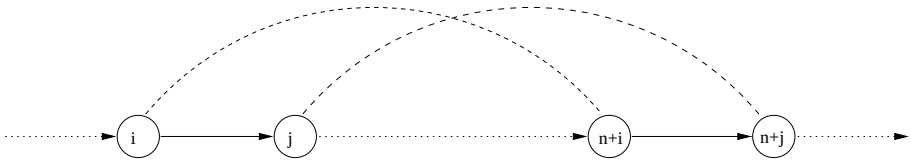

Figure 1: Blocks $B_k(i, n+i)$ and $B_k(j, n+j)$ overlap

3.2 Insertion operators

The proposed methodology is based on a partial destruction of the current solution at each iteration and its reconstruction with an insertion operator. In the latter case, all feasible insertion places of a given request must be considered in order to identify the best one. This procedure thus needs to be efficient. Fortunately, the LIFO constraint imposes a particular structure on the route, which can be exploited by the insertion procedure.

Extending the terminology in [3], if we assume that a given request i is put in stack k then a block $B_k(i, n+i)$ in the current route is the path from the pickup to the delivery location of customer request i. The block $B_k(i, n+i)$ is *simple* if there is no block $B_k(j, n+j)$ between i and $n+i$. It is *composed* if $B_k(i, n+i)$ contains one ore more *subblocks* $B_k(j, n+j)$. Two customer requests i and j in stack k satisfies the LIFO constraint if $B_k(i, n+i)$ and $B_k(j, n+j)$ do not have any vertex in common or if one block is a subblock of the other. Otherwise, the two blocks overlap and violate the LIFO constraint (see Figure 1).

Accordingly, after the insertion of pickup location i, only a restricted number of insertion places for the delivery location $n+i$ satisfies the LIFO constraint. These insertion places are identified as follows. First, the position just after i is clearly feasible for $n+i$. Then, the route after i is swept as follow:

- if the vertex is not in the same stack than i, then it is possible to insert $n+i$ just after that vertex;
- If the vertex is in the same stack k than i and is a pickup j, the block $B_k(j, n+j)$ is jumped and the search restarts from $n+j$. That is, if $n+i$ is inserted within $B_k(j, n+j)$, the two blocks $B_k(i, n+i)$ and $B_k(j, n+j)$ would violate the LIFO constraint.
- if the vertex is in the same stack k than i and is a delivery $n+j$, then
  - if j is after i in the route, $n+j$ has been reached by jumping from j. Hence, it is possible to insert $n+i$ just after $n+j$.
  - if j is before i in the route, there is no other feasible location along the route. Clearly, the insertion of $n+i$ at any place after $n+j$ would violate the LIFO constraint.

The two insertion operators will now be described.

3.2.1 Least-cost insertion

Here, the next request is inserted at the feasible location that incurs the smallest additional cost. This is the detour, in our case, since the objective is to minimize the total distance. More precisely if the pickup i is inserted between vertex j and its immediate successor $succ(j)$ and the delivery $n+i$ between l and $succ(l)$, the detour δ is:

$$\delta_{i,j,l} = (c_{j,i} + c_{i,succ(j)} - c_{j,succ(j)}) + (c_{l,n+i} + c_{n+i,succ(l)} - c_{l,succ(l)}) \tag{16}$$

It should be noted that the requests are reinserted one by one based on their removal order.

3.2.2 Regret-based insertion

An alternate insertion heuristic has been designed that alleviates the myopic behavior of the previous insertion heuristic. This is done through a variable reinsertion order based on a regret measure. The classical regret considers the difference between the cost of the second best and best insertion places in the solution. If this difference is large, the corresponding request gets high priority because a large cost is incurred if its best insertion place becomes infeasible (due to the insertion of other requests). A generalized variant for a multi-vehicle routing problem [13] considers the best feasible insertion place in each route and sums up the differences, over all routes, between the best insertion cost in the route and the best overall insertion cost. Here, this idea is exploited by considering stacks instead of routes. Namely, the impact on the route of each (previously) removed request is evaluated by considering its addition at every feasible position in each stack, in order to identify its best position in each stack. Then, its generalized regret measure is calculated.

Let us assume that the minimum detour in the route when request i is put in stack $k \in M$ is $\delta'_{i,k}$ and that the overall minimum detour is obtained when the request is put in stack k^*. Then, the generalized regret measure r_i is:

$$r_i = \sum_{k=1,\ldots,m} \left(\delta'_{i,k} - \delta'_{i,k^*}\right). \tag{17}$$

The next request chosen for reinsertion is the one with the largest generalized regret. Obviously, this request is put at the best position in its best stack, namely the one which leads to the smallest detour in the route. It should be noted that a classical 2-regret insertion heuristic is also available, where only the difference between the second-best stack and the best stack is considered. For problems with only one stack, the regret heuristic is based on the difference between the second best and best positions in the same stack.

3.3 Algorithmic framework

The general search scheme of LNS can now be described as follows, where S and $S*$ denote the current and best solution, respectively, and f is the objective function:

1. Create an initial solution S ;
2. $S^* \leftarrow S$;
3. $iter \leftarrow 1$;
4. While $iter \leq iter_{max}$ do
   - 4.1 Select a number of requests $1 \leq q \leq n$;
   - 4.2 Select a removal operator and an insertion operator ;
   - 4.3 Apply the removal and insertion operators to solution S to obtain S' ;
   - 4.4 If $f(S') < f(S)$ then
     - 4.4.1 $S \leftarrow S'$;
     - 4.4.2 If $f(S') < f(S^*)$ then $S^* \leftarrow S'$;
   - 4.5 If $f(S') \geq f(S)$ then
     - $S \leftarrow S'$ according to some acceptance criterion ;
   - 4.6 $iter \leftarrow iter + 1$;
5. Return S^*.

LNS is a rather simple iterative algorithm, which is applied for a fixed number of iterations $iter_{max}$ from an initial solution obtained with the least-cost insertion heuristic presented in section 3.2.1. The acceptance criterion in step 4.5, when solution S' does not improve S, is the one used in simulated annealing [11]. That is, S' is accepted with probability $e^{-\frac{f(S')-f(S)}{T}}$ where T is the temperature parameter. Starting from some initial value, the temperature is reduced from one iteration to the next by setting $T \leftarrow \alpha T$. Clearly, the probability of accepting a non improving solution diminishes with the value of T, as the algorithm unfolds. This behavior allows the algorithm to progressively settle in a (hopefully) good local optimum. In our experiments, the starting temperature was set to $1.05 f(S_0)$, where S_0 is the initial solution, and α to 0.99975, as suggested in [14].

3.4 Postprocessing

An exact dynamic programming algorithm is applied at the end to find the optimal route based on the evolution of the stacks observed in the solution returned by LNS. To this end, the evolution of each stack is represented by a feasible sequence of pickup and delivery vertices. For example, [1, n+1, 2, n+2, 0] indicates that the demand at pickup location 1 is successively loaded and unloaded. Then, the same applies to the demand at pickup location 2. A different evolution can be represented by [1, 2, n+1, n+2, 0] to indicate that the two demands are first loaded before being unloaded. It should be noted that the depot 0 is always the last vertex in this representation.

We have:

- $M = \{M_1, ..., M_k, ..., M_m\}$, the set of feasible sequences of pickups and deliveries associated with each stack (see the representation above);
- M_k, the sequence of pickups and deliveries associated with stack k, $1 \leq k \leq m$;
- $T = \{t_1, ..., t_k, ..., t_m\}$ the set of current positions in the sequences of pickups and deliveries;
- M_{k,t_k}, the pickup or delivery vertex at position t_k in M_k;
- $h_k(T)$ a function that increments the current position in stack k; that is, T becomes $\{t_1, ..., t_k + 1, ..., t_m\}$.

The recurrence relation is then written as follows:

$f^*(x, T) = \min_{k \in M, M_{k,t_k} \neq 0}\{c_{x,M_{k,t_k}} + f^*(M_{k,t_k}, h_k(T))\}$ if $M_{k,t_k} \neq 0$ for at least one k, $1 \leq k \leq m$,

$f^*(x, T) = c_{x0}$ otherwise

At the start, the current route is empty, vertex x is the depot 0 and T indicates the first position in each sequence. At each step, a vertex at one of the positions indicated by T is selected and inserted at the end of the current route. The recurrence stops when T indicates the last position in each sequence, which corresponds to the depot 0.

4 Computational Results

The experiments reported in this section have been performed on a 2.2 GHz AMD Opteron 275. In the following, the generation of our 1-PDPMS test instances is first described. Then, different sensitivity analysis experiments are reported, involving the number of requests q to be removed and the various removal and insertion operators. The final results obtained with LNS are then reported. This is followed by a comparison of LNS with state-of-the-art methods on benchmark instances for the TSPPDL and DTSPMS, which are special cases of the 1-PDPMS.

4.1 1-PDPMS instances

Our 1-PDPMS instances are derived from those for the TSPPDL in [3]. These instances contain between 25 and 751 vertices (including the depot). Three different classes were designed: class $C1$ adds a second stack to the original instances, while keeping an infinite capacity for each stack; class $C2$ contains instances with 2 to 5 stacks, a unit demand for every customer request and a capacity constraint on each stack; class $C3$ also contains instances with 2 to 5 stacks, but the demand varies between 1 and 10 units and the capacity constraint is tighter. Clearly, the degree of difficulty increases from class $C1$ to $C3$. The characteristics of these instances are

Identifier	*Size*	*C1*		*C2*		*C3*	
		# Stacks	*Cap.*	*# Stacks*	*Cap.*	*# Stacks*	*Cap.*
brd14051	25	2	∞	3	2	3	16
	51	2	∞	3	3	3	14
	75	2	∞	3	4	3	18
	101	2	∞	4	2	4	10
	251	2	∞	3	14	3	28
	501	2	∞	3	20	3	35
	751	2	∞	2	25	2	35
d15112	25	2	∞	3	2	3	12
	51	2	∞	4	1	4	10
	75	2	∞	3	4	3	18
	101	2	∞	2	6	2	22
	251	2	∞	2	14	2	28
	501	2	∞	3	20	3	35
	751	2	∞	3	25	3	35
d18512	25	2	∞	3	2	3	12
	51	2	∞	5	2	5	10
	75	2	∞	2	4	2	18
	101	2	∞	4	1	4	10
	251	2	∞	3	14	3	28
	501	2	∞	2	5	2	18
	751	2	∞	3	25	3	35
fnl4461	25	2	∞	3	2	3	12
	51	2	∞	3	3	3	14
	75	2	∞	5	2	5	13
	101	2	∞	3	6	3	22
	251	2	∞	4	5	4	19
	501	2	∞	2	20	2	35
	751	2	∞	3	25	3	35
nrw1379	25	2	∞	2	1	2	10
	51	2	∞	4	2	4	10
	75	2	∞	3	8	3	10
	101	2	∞	2	11	2	10
	251	2	∞	2	6	2	10
	501	2	∞	2	10	2	10
	751	2	∞	3	15	3	10
pr1002	25	2	∞	3	1	3	10
	51	2	∞	3	2	3	14
	75	2	∞	4	2	4	10
	101	2	∞	3	4	3	20
	251	2	∞	3	4	3	14
	501	2	∞	3	8	3	25
	751	2	∞	2	8	2	25

Table 1: $1 - PDPMS$ instances

shown in Table 1. The identifier, number of vertices, number of stacks and capacity for each class, are shown in this order.

In the following sections, a number of sensitivity analysis experiments are described. For these experiments, 16 instances with 251 vertices or less were selected over the three classes and 10 runs were executed on each instance, with 25,000 iterations per run (for a total of 250,000 iterations). The number of iterations was set large enough for the improvement curve to stabilize even on the largest instances.

4.2 Number of removed requests

The number of requests that are removed from the current solution has a clear impact on the computation time, but its impact on solution quality is not so clear. Different removal strategies have thus been devised to study this issue. These strategies use four non negative values n_{min}, n_{max}, fr_{min} and fr_{max}, which correspond to the minimum and maximum number of requests to be removed, represented either as an absolute number or a fraction (between 0 and 1) of the total number of requests. More precisely, the number q is randomly chosen at each iteration in the interval $[\min\{n \cdot fr_{min}, n_{min}\}, \min\{n \cdot fr_{max}, n_{max}\}]$, where $n \cdot fr_{min}$ and $n \cdot fr_{max}$ are rounded to the nearest integer. The four strategies can now be described as follows.

- strategy S_1 sets $n_{min} = n_{max} = \infty$ and $fr_{min} = fr_{max}$. A fixed number of requests is thus removed at each iteration.
- strategy S_2 sets $n_{min} = 1$, $n_{max} = \infty$ and $fr_{min} = 1$. Here, the minimum number of requests is 1 and the maximum is determined by fr_{max}. Thus, a variable number of requests is removed at each iteration.
- strategy S_3 sets $n_{min} = n_{max} = \infty$, while fr_{min} and fr_{max} are set to different values such that $fr_{min} < fr_{max}$. Thus, the interval of admissible values for q is defined through fr_{min} and fr_{max} for which the lower bound is typically larger than 1 (as opposed to S_2).
- strategy S_4 does not impose any specific values as long as $n_{min} < n_{max}$ and $fr_{min} < fr_{max}$. This approach avoids the removal of too few requests in the case of small instances or too many requests in the case of large instances, by explicitly setting an absolute minimum and maximum number of requests.

We have implemented the four strategies with different values for n_{min}, n_{max}, fr_{min} and fr_{max}. We do not show the detailed results here. Only the best results, which were all obtained with S_4, are presented in Table 2. In this table, the solution quality corresponds to the gap in percentage of the average solutions obtained with a given interval over the average of the best known solutions for the 16 test instances. The average computation time in seconds for a single run is also reported.

Strategy S_4 is quite robust for the intervals reported in the table, with a gap varying only between 1.62% and 1.94%. The first interval, which is the least computationally expensive, was finally chosen for the next experiments.

Test	*Interval*	*Sol (%)*	*CPU (s)*
1	$[min\{10, 0.15n\}, min\{35, 0.45n\}]$	1.85	30.6
2	$[min\{15, 0.15n\}, min\{40, 0.45n\}]$	1.72	33.0
3	$[min\{20, 0.15n\}, min\{45, 0.45n\}]$	1.93	35.0
4	$[min\{30, 0.15n\}, min\{50, 0.45n\}]$	1.86	36.1
5	$[min\{10, 0.20n\}, min\{35, 0.55n\}]$	1.76	41.2
6	$[min\{15, 0.20n\}, min\{40, 0.55n\}]$	1.80	46.0
7	$[min\{20, 0.20n\}, min\{45, 0.55n\}]$	1.94	48.9
8	$[min\{30, 0.20n\}, min\{50, 0.55n\}]$	1.62	51.1
9	$[min\{10, 0.30n\}, min\{35, 0.60n\}]$	1.82	53.9
10	$[min\{15, 0.30n\}, min\{40, 0.60n\}]$	1.69	55.0
11	$[min\{20, 0.30n\}, min\{45, 0.60n\}]$	1.75	60.0
12	$[min\{30, 0.30n\}, min\{50, 0.60n\}]$	1.77	64.0

Table 2: Results of strategy S_4 with different intervals

Insertion	*Removal*	*Sol (%)*	*CPU (s)*
Least-cost	Random	2.59	5.3
	Distance	2.54	9.0
	Route	3.67	5.1
	Stack	2.31	8.9
	All	2.28	6.9
	All minus Route	2.55	7.1
Regret	Random	1.79	51.5
	Distance	1.81	54.4
	Route	2.94	49.6
	Stack	1.77	55.1
	All	1.63	53.0
	All minus Route	1.73	53.2
All	Random	1.84	42.5
	Distance	1.69	45.8
	Route	3.09	41.6
	Stack	1.73	46.0
	All	1.65	43.7
	All minus Route	1.66	43.9

Table 3: Results with different subsets of operators

4.3 Operators

This subsection studies the impact of the various removal and insertion operators on solution quality. To this end, we consider variants of LNS using only a subset of operators. As indicated in Table 3, implementations based on only one insertion operator, either Least-Cost or Regret, or both insertion operators are tested in combination with only one removal operator, with all removal operators or with all removal operators minus the Route-based operator, which proved to be the worst.

Although the Route-based removal operator is much worse than the other removal operators, its inclusion seems to be beneficial (perhaps, by providing a form of diversification), as solution quality degrades when it is discarded. The Stack-based removal operator alone is quite good and performs well overall when compared with the other implementations based on a single removal operator. The table also in-

Class	*Size*	*LNS*		
		Best (%)	*Avg (%)*	*CPU (s)*
C1	25	0.00	0.00	0.4
	51	0.00	0.01	2.9
	75	0.00	0.57	9.4
	101	0.04	0.19	25.1
	251	0.10	1.11	184.4
	501	0.00	2.43	519.2
	751	0.85	2.77	1112.5
C2	25	0.00	0.00	0.4
	51	0.00	0.09	2.9
	75	0.00	0.50	10.3
	101	0.00	0.46	26.8
	251	0.00	1.33	182.4
	501	3.00	4.15	548.0
	751	2.75	3.56	1126.9
C3	25	0.00	0.00	0.4
	51	0.00	0.19	2.8
	75	0.00	0.93	9.0
	101	0.13	0.74	25.3
	251	0.44	3.53	178.0
	501	0.51	2.47	517.0
	751	1.43	3.38	938.4

Table 4: Results on 1-PDPMS instances

dicates that the Regret-based insertion outperforms the Least-cost one, but is also more computationally expensive. When all insertion operators are combined with all removal operators, a very small degradation in solution quality is observed, as compared with the use of Regret-based insertion alone (0.02%), but the CPU time is significantly reduced. Based on these observations, all operators were kept in the final implementation.

Finally, it is worth noting that the postoptimization with the dynamic programming algorithm allows a further average improvement of 0.2% in solution quality in only 0.04 second of computation time.

4.4 1-PDPMS

The results obtained with the final implementation of LNS on the full test set are shown in Table 4, based on 10 runs on each instance with 25,000 iterations for each run. The results are summarized by taking averages over instances of the same size in classes $C1$, $C2$ and $C3$. For each class and size, the following numbers are reported: the average of the best solutions, the average solution and the average CPU time in seconds (for a single run). Solution quality is reported as a gap in percentage over the average of the best known solutions for the corresponding class and size. The best known solutions have been obtained by running different variants of LNS for large run times.

Note that the instances in class $C1$ have been created by adding a second stack of infinite capacity to the original instances in [3]. By comparing the results on

class $C1$ with those reported in [3], we observed an average improvement of 22% in total distance. This is not a surprise, given the additional flexibility provided by the second stack.

4.5 TSPPDL

Benchmark instances for the TSPPDL, which is a special case of the 1-PDPMS when the vehicle contains a single stack of infinite capacity, are found in [2, 3]. The 63 instances in [2] are small because they were designed to test an exact algorithm. This exact algorithm found the optimum on 52 of these instances. A second test set of 42 instances is reported in [3]. These instances range in size from 25 to 751 vertices and were designed to test a Variable Neighborhood Search (VNS).

The results on the first test set are shown in Table 5 in the usual format. When the optimum is not known for a given instance, the best known solution, as reported in [2], is shown in italic. This table shows that LNS found the optimum or best known in all cases but one, when the best of 10 runs is considered. It also improved the best known solution for the instance nrw1379 with 35 vertices. Even when the average of 10 runs is considered, the solutions obtained are only 0.06% over those reported in [2] with CPU times that do not exceed 1 second.

The results on the larger instances in the second test set are found in Table 6, including those obtained with the VNS heuristic [3]. In the latter case, the authors report the average solution values over 10 runs on each instance. As usual, the gap in percentage over the best known solution is shown, where the best solution has been obtained by running different variants of our LNS algorithm on each instance for large run times. It should also be noted that the CPU times of VNS should be halved to account for the greater speed of our machine. With regard to the average over 10 runs, LNS provides significant improvements over VNS (i.e., 0.49% versus 2.6%) in less CPU time (even after halving it for VNS). By taking the best of 10 runs, LNS can also find solutions that are only 0.1% over the best known, on average.

4.6 DTSPMS

The 1-PDPMS can be transformed into a DTSPMS by modifying the cost matrix to force a pickup route to be followed by a delivery route. Let us assume that the pickup and delivery routes start and end at depots 0_p and 0_d, respectively, and that the travel cost between 0_p and 0_d is ignored. Then, we have:

$$c_{i,j} = c_{i,0_p} + c_{0_d,j},\ c_{j,i} = c_{0_p,j} = c_{i,0_d} = \infty,\ i \in P,\ j \in D.$$

That is, the travel cost between a pickup and a delivery vertex is equal to the travel cost from the pickup to 0_p and from 0_d to the delivery. Furthermore, the travel cost from a delivery to a pickup vertex is set to infinity to force all pickups to be performed before the deliveries. Finally, the travel cost between two pickups or two deliveries remains the same.

Instance	*Size*	*Optimum*	*LNS*		
			Best (%)	*Avg (%)*	*CPU (s)*
a280	19	402	0.00	0.00	0.2
	23	468	0.00	0.00	0.2
	27	505	0.00	0.00	0.3
	31	560	0.00	0.00	0.4
	35	647	0.00	0.00	0.6
	39	691	0.00	0.00	0.8
	43	752	0.00	0.00	1.0
att532	19	4250	0.00	0.00	0.2
	23	5038	0.00	0.00	0.2
	27	5800	0.00	0.00	0.3
	31	6173	0.00	0.00	0.4
	35	6361	0.00	0.00	0.6
	39	6725	0.00	0.00	0.8
	43	*10714*	0.00	0.05	1.0
brd14051	19	4555	0.00	0.00	0.2
	23	4655	1.29	1.29	0.2
	27	4936	0.00	0.00	0.3
	31	5186	0.00	0.00	0.4
	35	5196	0.00	0.00	0.6
	39	*5629*	0.00	0.00	0.7
	43	*5719*	0.00	0.00	1.0
d15112	19	76203	0.00	0.00	0.2
	23	88272	0.00	0.16	0.2
	27	93158	0.00	0.25	0.3
	31	109166	0.00	0.22	0.5
	35	115554	0.00	0.00	0.6
	39	*119863*	0.00	0.00	0.8
	43	*128798*	0.00	0.00	1.0
d18512	19	4446	0.00	0.00	0.2
	23	4658	0.00	0.00	0.2
	27	4704	0.00	0.00	0.3
	31	5120	0.00	0.00	0.5
	35	5186	0.00	0.00	0.6
	39	*5419*	0.00	0.00	0.7
	43	*5634*	0.00	0.00	1.0
fnl4461	19	1866	0.00	0.00	0.2
	23	2067	0.00	0.00	0.2
	27	2483	0.00	0.00	0.3
	31	2672	0.00	0.00	0.4
	35	2852	0.00	0.00	0.6
	39	3109	0.00	0.00	0.7
	43	*3269*	0.00	0.00	1.0

Instance	*Size*	*Optimum*	*LNS*		
			Best (%)	*Avg (%)*	*CPU (s)*
nrw1379	19	2691	0.00	0.00	0.2
	23	2919	0.00	0.00	0.2
	27	3366	0.00	0.00	0.3
	31	3554	0.00	0.00	0.5
	35	*3652*	-0.22	-0.22	0.6
	39	*4002*	0.00	0.00	0.8
	43	*4282*	0.00	0.00	1.0
pr1002	19	12947	0.00	0.00	0.2
	23	13872	0.00	0.00	0.2
	27	15566	0.00	0.00	0.3
	31	16255	0.00	0.00	0.4
	35	17564	0.00	0.00	0.6
	39	18862	0.00	0.00	0.7
	43	20173	0.00	0.00	1.0
ts225	19	21000	0.00	0.00	0.2
	23	25000	0.00	0.00	0.2
	27	32395	0.00	0.00	0.3
	31	33395	0.00	1.65	0.5
	35	36703	0.00	0.13	0.6
	39	39395	0.00	0.00	0.8
	43	43082	0.00	0.00	1.0
Avg			0.02	0.06	0.5

Table 5: Results on the first test set of Carrabs, Cerruli and Cordeau

A test set with 60 randomly generated Euclidean DTSPMS instances is reported in [12], where the distances have been rounded to the nearest integer. More precisely, there are 20 instances with 12, 33 and 66 customer requests (i.e, 26, 68 and 132 vertices, including the two depots). All customer requests have a unit demand. In each instance, the vehicle contains three stacks and the capacity of each stack is one third of the total demand. The optimal solutions are known for the instances with 12 requests, but for the larger ones, the best known solutions have been obtained by Petersen and Madsen [12] after multiple 2-hour runs of their algorithm on each instance. Solution quality is thus represented as the gap in percentage with either the optimal solutions or the best known solutions produced by Petersen and Madsen.

In Tables 7, 8 and 9, the solutions obtained with our algorithms are compared with the solutions of Petersen and Madsen for the instances with 12, 33 and 66 requests, respectively. The algorithm of Madsen and Petersen is a combination of a large neighborhood search (using insertion and removal operators different from ours) and a local search based on vertex exchanges. In the tables, *Short* and *Long* refer to two different calibrations of this algorithm for short (10 seconds) and long (3 minutes) runs. It should be noted that the machine and programming language used in our implementation lead to running times that are about 3 times faster than those of Petersen and Madsen. Accordingly, *Short* and *Long* would approximately run for 3 seconds and 1 minute, respectively, on our machine. The solutions reported by Felipe et al. in [4], obtained with a variable neighborhood search approach, are also shown in Tables 8 and 9 (the authors do not provide detailed results on each instance

Instance	*Size*	*VNS*		*LNS*		
		Avg (%)	*CPU (s)*	*Best (%)*	*Avg (%)*	*CPU (s)*
fnl4461	25	0.00	0.0	0.00	0.00	0.3
	51	0.00	0.1	0.00	0.00	1.5
	75	2.20	0.2	0.00	0.00	4.6
	101	3.78	0.7	0.00	0.32	10.5
	251	2.51	23.9	0.00	0.66	71.5
	501	3.95	458.6	0.00	1.29	215.1
	751	3.53	2172.5	0.00	0.57	532.8
brd14051	25	0.22	0.0	0.00	0.00	0.3
	51	0.30	0.1	0.00	0.00	1.5
	75	1.07	0.3	0.00	0.00	4.0
	101	2.82	0.7	0.00	0.01	10.2
	251	8.44	36.7	0.00	1.83	72.0
	501	5.56	478.7	0.89	1.68	213.2
	751	4.63	2169.8	0.75	1.32	482.8
d15112	25	0.00	0.0	0.00	0.00	0.3
	51	1.03	0.0	0.00	0.00	1.4
	75	1.20	0.2	0.00	0.00	4.5
	101	4.41	0.5	0.00	0.36	10.8
	251	4.80	24.6	0.00	0.94	73.0
	501	3.01	385.9	0.03	1.03	218.1
	751	2.22	1968.6	0.00	0.83	465.8
d18512	25	0.24	0.0	0.00	0.00	0.3
	51	0.85	0.1	0.00	0.00	1.4
	75	1.77	0.2	0.00	0.00	3.9
	101	0.88	0.5	0.00	0.00	10.2
	251	6.94	32.9	0.15	0.99	70.2
	501	5.65	486.0	0.00	0.71	243.4
	751	3.58	2508.5	0.51	0.86	576.2
nrw1379	25	0.09	0.0	0.00	0.00	0.3
	51	0.79	0.1	0.00	0.00	1.4
	75	0.50	0.2	0.00	0.00	4.5
	101	3.88	0.5	0.00	0.34	9.9
	251	5.54	24.5	0.00	0.89	72.3
	501	3.60	380.1	0.22	1.20	218.9
	751	2.23	2447.1	0.00	0.74	516.4
pr1002	25	0.00	0.0	0.00	0.00	0.3
	51	0.81	0.1	0.00	0.00	1.5
	75	0.67	0.3	0.21	0.22	4.4
	101	3.44	0.8	0.00	0.19	10.8
	251	5.86	31.3	1.19	1.87	68.6
	501	3.57	471.9	0.00	1.25	213.5
	751	2.80	2785.4	0.04	0.49	500.7
Avg		2.60	402.2	0.10	0.49	117.2

Table 6: Results on the second test set of Carrabs, Cordeau and Laporte

for the test set with 12 requests). Average solutions, over 3 runs, are reported after 10 seconds and 3 minutes of computation times. No CPU time adjustment is needed here, because the machine used by Felipe et al. is similar to ours.

Table 7 provides comparisons with the optimum on the small 12-request instances. In the case of Petersen and Madsen, the results are averages of three runs, and the optimum has been found in all cases. The average solution value of LNS is 0.08% over the optimum, but the best of 10 runs is always optimal. These 10 runs execute in $10 \cdot 0.5 = 5$ seconds which is comparable to the *Short* runs of Petersen and Madsen. It should be noted that Felipe et al. report an average gap (over three runs on each instance) of 0.2%, in only one second of CPU time. They did not find the optimum on two instances.

Tables 8 and 9 show the results for the larger instances with 33 and 66 customer requests. On the instances with 33 requests, the average run of LNS is 0.65% over the best known using only 11.1 seconds of CPU time. This is an improvement over the (single) *Long* run of Petersen and Madsen which provides a solution which is 1% over the best known in 1 minute of equivalent CPU time. When the best of 10 runs is considered, LNS gets as close as 0.05% over the best known in $10 \cdot 11.1 = 111$ seconds. The results produced by LNS are very similar to those obtained by Felipe et al. who report average gaps of 0.65% after 10 seconds and 0.05% after 3 minutes. On the instances with 66 requests, LNS is 2.76% over the best known on average in less than 2 minutes of CPU time. This is better than the (single) *Long* run of Petersen and Madsen which is 8% over the best known, admittedly after only 1 minute of equivalent CPU time. By reducing the number of iterations of LNS from 25,000 to 12,500, to obtain runs of approximately 1 minute, the average solution of LNS remains at 3.28% over the best known, which is still better than Petersen and Madsen. The average gap of 2.76% in less than 2 minutes is also better than the *Long* run of Felipe et al. which is 3.2% over the best known after 3 minutes of CPU time. When the best of 10 runs is considered, LNS gets solutions that are 1.17% over the best known, on average. Furthermore, a best known solution was found on instance R05-66.

Finally, Felipe et al. [4] have produced a test set of 20 instances with 132 requests, which were generated like the previous instances. They report results obtained with their algorithm after 10 seconds, 3 minutes, 5 minutes and 8 minutes of computation time. They are reported in Table 10 with the results of LNS. The best solutions of Felipe et al. have been obtained by running their algorithm for 12 hours on each instance. As observed in this table, the average run of LNS is 1.64% over the best solutions produced by Felipe et al., after less than 7 minutes. This is to be compared with an average gap of 3.4% for Felipe et al. after 8 minutes. Furthermore, when the best of 10 runs is considered (for a total computation time slightly larger than one hour), LNS has found 12 new best solutions out of 20 and the average of these solution values is better than the average of the best solutions produced by Felipe et al. after 12 hours of computation time. It thus seems that the difference in performance between LNS and the VNS approach of Felipe et al. increases with problem size.

Instance	*Petersen & Madsen*			*LNS*		
	Optimal	*Short (%)*	*Long (%)*	*Best (%)*	*Avg (%)*	*CPU (s)*
R00-12	694	0	0	0.00	0.52	0.5
R01-12	710	0	0	0.00	0.31	0.5
R02-12	606	0	0	0.00	0.00	0.5
R03-12	680	0	0	0.00	0.00	0.5
R04-12	607	0	0	0.00	0.00	0.5
R05-12	567	0	0	0.00	0.00	0.5
R06-12	747	0	0	0.00	0.00	0.5
R07-12	557	0	0	0.00	0.00	0.5
R08-12	690	0	0	0.00	0.00	0.5
R09-12	669	0	0	0.00	0.00	0.5
R10-12	633	0	0	0.00	0.00	0.5
R11-12	591	0	0	0.00	0.00	0.5
R12-12	722	0	0	0.00	0.17	0.5
R13-12	664	0	0	0.00	0.00	0.5
R14-12	650	0	0	0.00	0.26	0.5
R15-12	595	0	0	0.00	0.27	0.5
R16-12	577	0	0	0.00	0.00	0.5
R17-12	737	0	0	0.00	0.00	0.5
R18-12	724	0	0	0.00	0.01	0.5
R19-12	753	0	0	0.00	0.16	0.5
Avg		0	0	0.00	0.08	0.5

Table 7: Results on the DTSPMS instances of Petersen and Madsen with 12 requests

Instance	*Petersen & Madsen*			*Felipe et al.*		*LNS*		
	Best	*Short (%)*	*Long (%)*	*Short (%)*	*Long (%)*	*Best (%)*	*Avg (%)*	*CPU (s)*
R00	1063	4	1	0.8	0.0	0.00	0.57	10.6
R01	1032	4	1	0.8	0.0	0.00	0.24	10.7
R02	1065	4	1	0.8	0.0	0.00	0.25	10.7
R03	1100	6	1	0.0	0.0	0.00	1.00	11.2
R04	1052	5	2	0.5	0.0	0.00	1.14	11.2
R05	1008	3	1	2.2	0.0	0.00	0.84	11.0
R06	1110	6	2	0.0	0.0	0.00	0.31	11.2
R07	1105	5	1	0.4	0.0	0.36	0.90	11.1
R08	1109	4	1	0.0	0.0	0.00	0.55	11.1
R09	1091	4	1	0.0	0.0	0.00	0.54	11.1
R10	1016	5	0	0.0	0.0	0.00	0.00	11.3
R11	1001	6	1	0.0	0.0	0.00	0.60	11.1
R12	1109	4	1	0.2	0.0	0.18	0.74	11.1
R13	1084	4	1	0.0	0.0	0.00	0.62	11.0
R14	1034	3	0	1.7	0.0	0.00	0.69	11.1
R15	1142	4	1	1.4	0.0	0.26	1.02	11.3
R16	1093	2	0	0.0	0.0	0.00	0.15	11.2
R17	1073	4	0	0.9	0.0	0.00	0.57	10.9
R18	1118	5	1	2.8	0.7	0.00	1.33	11.3
R19	1089	3	1	0.6	0.2	0.28	0.91	11.2
Avg		4	1	0.66	0.05	0.05	0.65	11.1

Table 8: Results on the DTSPMS instances of Petersen and Madsen with 33 requests

Instance	*Petersen & Madsen*			*Felipe et al.*		*LNS*		
	Best	*Short (%)*	*Long (%)*	*Short (%)*	*Long (%)*	*Best (%)*	*Avg (%)*	*CPU (s)*
R00-66	1594	19	7	3.8	3.1	0.31	2.65	108.9
R01-66	1600	20	8	6.4	4.3	1.25	3.09	110.1
R02-66	1576	20	12	7.7	6.2	3.55	4.90	109.6
R03-66	1631	14	6	5.9	2.5	0.67	2.02	111.3
R04-66	1611	18	9	7.7	2.9	1.49	3.16	111.4
R05-66	1528	18	7	6.9	3.3	-0.13	1.30	112.1
R06-66	1651	17	7	10.5	3.6	0.42	2.13	111.5
R07-66	1653	17	8	6.4	1.0	1.21	3.02	111.6
R08-66	1607	18	7	11.1	3.4	0.62	3.04	111.6
R09-66	1598	18	8	8.6	3.1	2.07	2.78	112.1
R10-66	1702	17	9	7.8	3.9	0.59	1.77	112.4
R11-66	1575	19	8	6.7	5.3	0.44	2.98	111.9
R12-66	1652	19	10	6.0	2.2	0.30	1.96	112.4
R13-66	1617	19	10	8.7	2.5	1.98	3.21	111.6
R14-66	1611	21	9	6.6	1.4	0.56	2.59	111.7
R15-66	1608	19	10	6.5	1.9	1.31	2.20	111.2
R16-66	1725	16	7	8.2	2.6	0.87	2.07	111.8
R17-66	1627	21	10	7.5	5.1	2.03	3.34	112.0
R18-66	1671	18	8	6.5	2.3	1.97	3.88	112.5
R19-66	1635	17	9	5.2	2.9	1.83	3.13	112.5
Avg		18	8	7.2	3.2	1.17	2.76	111.5

Table 9: Results on the DTSPMS instances of Petersen and Madsen with 66 requests

Instance	*Felipe et al.*					*LNS*			
	Best	*10s (%)*	*3min (%)*	*5min (%)*	*8min (%)*	*Best*	*Best (%)*	*Avg (%)*	*CPU (s)*
R00-132	2591	15.7	6.6	6.3	3.8	2590	-0.04	2.57	400.4
R01-132	2645	16.7	5.4	4.2	4.5	2650	0.19	2.14	403.8
R02-132	2639	13.9	5.2	4.6	3.1	2679	1.52	2.97	401.1
R03-132	2752	12.4	3.2	4.2	1.7	2698	-1.96	0.65	398.7
R04-132	2603	13.1	4.6	3.0	2.9	2590	-0.50	0.41	405.7
R05-132	2616	15.8	5.7	4.7	4.7	2651	1.34	2.27	401.2
R06-132	2576	16.0	7.1	4.6	4.9	2579	0.12	1.93	403.2
R07-132	2615	14.7	7.5	4.8	3.6	2559	-2.14	1.00	401.2
R08-132	2638	14.3	5.3	4.6	3.4	2636	-0.08	1.71	402.8
R09-132	2554	13.6	3.5	2.5	1.3	2499	-2.15	-0.50	399.8
R10-132	2646	19.0	6.4	3.7	3.6	2663	0.64	1.92	400.1
R11-132	2632	13.3	4.6	5.2	2.7	2621	-0.42	1.77	401.8
R12-132	2555	18.5	6.8	5.3	5.4	2544	-0.43	2.56	401.5
R13-132	2659	15.7	5.0	3.3	2.1	2664	0.19	1.76	404.2
R14-132	2605	14.0	3.7	2.5	2.7	2568	-1.42	1.13	403.5
R15-132	2626	18.5	5.3	5.4	3.8	2634	0.30	1.75	402.1
R16-132	2534	16.0	6.7	4.5	4.6	2585	2.01	3.30	400.4
R17-132	2569	14.2	4.6	4.3	3.5	2559	-0.39	1.58	402.7
R18-132	2652	15.1	3.5	2.0	1.9	2628	-0.90	0.66	402.3
R19-132	2644	16.0	4.2	4.6	2.8	2624	-0.76	1.31	401.6
Avg		15.3	5.3	4.2	3.4		-0.24	1.64	401.9

Table 10: Results on the DTSPMS instances of Felipe et al. with 132 requests

5 Conclusion

This paper has described a large neighborhood search which is both efficient and effective for solving the 1-PDPMS. This is also true for special cases of this problem, like the TSPPDL and DTSPMS, as empirically demonstrated through comparisons with state-of-the-art methods on different sets of benchmark instances. Building on these results, our aim is now to address the more challenging multi-vehicle extension of the problem.

Acknowledgments. Financial support for this work was provided by the Canadian Natural Sciences and Engineering Research Council (NSERC). This support is gratefully acknowledged.

References

[1] Ahuja R.K., Ergun Ö., Orlin J.B., Punnen A.P., "A Survey of Very Large-Scale Neighborhood Search Techniques", *Discrete Applied Mathematics 123*, 75–102, 2002.

[2] Carrabs F., Cerulli R., Cordeau J.-F., "An Additive Branch-and-Bound Algorithm for the Pickup and Delivery Traveling Salesman Problem with LIFO or FIFO Loading", *INFOR 45*, 223-238, 2007.

[3] Carrabs F., Cordeau J.-F., Laporte G., "Variable Neighborhood Search for the Pickup and Delivery Traveling Salesman Problem with LIFO Loading", *INFORMS Journal on Computing 19*, 618-632, 2007.

[4] Felipe A., Ortuno M.T., Tirado G., "The Double Traveling Salesman Problem with Multiple Stacks: A Variable Neighborhood Search Approach", *Computers & Operations Research 36*, 2983-2993, 2009.

[5] Doerner K.F., Fuellerer G., Gronalt M., Hartl R.F., Iori M., "Metaheuristics for the Vehicle Routing Problem with Loading Constraints", *Networks 49*, 294–307, 2007.

[6] Fuellerer G., Doerner K.F., Hartl R.F., Iori M., "Ant Colony Optimization for the Two-Dimensional Loading Vehicle Routing Problem", *Computers & Operations Research 36*, 655-673, 2009.

[7] Fuellerer G., Doerner K.F., Hartl R.F., Iori M., "Metaheuristics for Vehicle Routing Problems with Three-Dimensional Loading Constraints", *European Journal of Operational Research*, forthcoming, 2009.

[8] Gendreau M., Iori M., Laporte G., Martello S., "A Tabu Search Algorithm for a Routing and Container Loading Problem", *Transportation Science 40*, 342-350, 2006.

[9] Gendreau M., Iori M., Laporte G., Martello S., "A Tabu Search Heuristic for the Vehicle Routing Problem with Two-Dimensional Loading Constraints", *Networks 51*, 4-18, 2008.

[10] Iori M., Salazar-González J.-J., Vigo D., "An Exact Approach for the Vehicle Routing Problem with Two-Dimensional Loading Constraints", *Transportation Science 41*, 253–264, 2007.

[11] Kirkpatrick S., Gelatt Jr. C.D., Vecchi M.P., "Optimization by Simulated Annealing", *Science 220*, 671–680, 1983.

[12] Petersen H.L., Madsen O.B.G., "The Double Traveling Salesman Problem with Multiple Stacks - Formulation and Heuristic Solution Approaches", *European Journal of Operational Research 198*, 139–147, 2009.

[13] Potvin J.-Y., Rousseau J.-M., "An Exchange Heuristic for Routing Problems with Time Windows", *Journal of the Operational Research Society 46*, 1433–1446, 1995.

[14] Ropke S., Pisinger D., "An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows", *Transportation Science 40*, 455–472, 2006.

[15] Schrimpf G., Schneider J., Stamm-Wilbrandt H., Dueck G., "Record Breaking Optimization Results using the Ruin and Recreate Principle", *Journal of Computational Physics 159*, 139–171, 2000.

[16] Shaw P., "Using Constraint Programming and Local Search Methods to solve Vehicle Routing Problems", *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming*, M. Maher and J.-F. Puget eds., Lecture Notes in Computer Science 1520, Springer-Verlag, Berlin, 417–431, 1998.