Heuristic Solution Approaches to the Double TSP with Multiple Stacks

Hanne L. Petersen
hlp@ctt.dtu.dk

May 24, 2006

Technical Report

Centre for Traffic and Transport
(CTT-TECHNICAL REPORT-2006-2)

Informatics and Mathematical Modelling
(IMM-TECHNICAL REPORT-2006-10)

Technical University of Denmark
DK-2800 Kgs. Lyngby, Denmark

Presented at:
Odysseus 2006, Third International Workshop
on Freight Transportation and Logistics
and
CORS / Optimization Days 2006

Abstract

This paper introduces the Double Travelling Salesman Problem with Multiple Stacks and presents a three different metaheuristic approaches to its solution.

The Double Travelling Salesman Problem with Multiple Stacks is concerned with finding the shortest route performing pickups and deliveries in two separated networks (one for pickups and one for deliveries). Repacking is not allowed, but the items can be packed in several *rows* in the container, such that each row can be considered a LIFO stack, but no mutual constraints exist between the rows.

Two different neighbourhood structures are developed for the problem and used with each of the heuristics. Finally some computational results are given along with lower bounds on the objective value.

1 Introduction

As congestion is an ever-growing problem on the roads of Europe, intermodality is playing an increasingly important role in the transportation of goods. Furthermore, the complexity of the planning problems thus induced presents additional requirements to the tools available to planners.

This project was initiated in cooperation with a company producing computer software systems for the operation and fleet management in small and medium-sized transportation companies. The software company presented a problem that is intriguing in that it does not seem to have been previously treated in the literature, but at the same time it is conceptually simple.

The Double Travelling Salesman Problem with Multiple Stacks (DTSPMS), is concerned with finding the shortest routes performing pickups and deliveries in two separated networks. The problem does not permit repacking, instead the items can be packed in several *rows* in the container, such that each row must obey the LIFO (Last-In-First-Out) principle, while there are no mutual constraints between the rows. Additionally no stacking is allowed.

In the DTSPMS a set of orders is given, each one requiring transportation from a given location in one region to a given location in another region, i.e. each order consists of a pickup location and a delivery location for one item. All items are required to be uniform. The two regions are far apart, and thus some long-haul transport is required between the depots in each of the regions. This long-haul transport is not part of the problem. All pickups and deliveries must be carried out with the same container, which cannot be repacked along the way. Hence the problem to be solved consists of the two geographically separated problems of picking up and delivering the items in a feasible way with regard to container loading. Items in the container can only be accessed from the opening in one end of the container, and the objective is to find a pair of tours with the shortest total length. No time windows are considered in this problem.

In practice this can occur when the container is loaded onto a truck that performs the pickups, then returned by that truck to a local depot where it is transferred onto a train or another truck, which then performs the long-haul transportation. Upon arrival at a depot in the delivery region, the container is again transferred to a truck which carries out the actual deliveries.

To state the problem more formally two weighted complete graphs $G^{\mathrm{P}} = (V^{\mathrm{P}}, E^{\mathrm{P}})$ and $G^{\mathrm{D}} = (V^{\mathrm{D}}, E^{\mathrm{D}})$ are given for pickup (P) and delivery (D) respectively, and the purpose is to find a Hamiltonian tour through each, such that the sum of the weights of the edges used is minimised. Each graph has a depot node v_0^τ, $\tau \in \{\mathrm{P}, \mathrm{D}\}$, and customer nodes $v_1^\tau, ..., v_n^\tau$. A set of n orders $R = \{1, \ldots, n\}$ is given, where order i must be picked up at $v_i^{\mathrm{P}} \in V^{\mathrm{P}}$

and delivered at $v_i^{\mathrm{D}} \in V^{\mathrm{D}}$. Finally, we let $V_C^\tau = V^\tau \backslash \{0\}$ denote the set of customer locations in τ.

The connection between the two tours to be found is given by the loading of the container. Since no repacking is allowed, the only items that can be delivered next at any time during delivery are the ones that are accessible from the opening of the container. This implies that the loading is subject to LIFO constraints.

However, in the DTSPMS there is no LIFO ordering for the container as a whole. Rather, it contains several loading rows, each of which can be considered a LIFO stack, but all rows are independently accessible.

In real life the items to be transported would typically be Euro Pallets, which fit 3 by 11 on the floor area of a 40-foot pallet container, providing three independent loading rows.

A solution to a given problem consists of a *pickup route*, a *delivery route*, and a *row assignment*, which for each item tells which loading row it should be placed in. A row assignment only gives the row that each item should be placed in, and does not indicate which position the item will occupy in that row. Given a route (pickup or delivery) and a row assignment, one can construct the *loading plan*, which gives the exact position of each item inside the container.

The problem may at first glance seem purely theoretical, since the extra mileage incurred by not being able to repack can seem prohibitive. However the problem has been encountered in real-life applications, where this extra mileage is justified by the wages stemming from handling and requirements to comply with union restrictions (the driver is not allowed to handle the goods).

Special cases of the problem occur when the number of loading rows is equal to one or to the number of orders n. In both cases the problem of finding a row assignment for the solution becomes irrelevant.

In the single row case the pickup route will strictly dictate the delivery route (or vice versa), and the two routes need to be exact opposites. In this case the problem can be solved by adding the distance matrices of the two graphs, and solving a regular TSP for the resulting distance matrix.

Conversely, when the number of loading rows equals the number of customers n, the two routes do not impose any restrictions on each other and the optimal solution to the DTSPMS consists of the optimal solutions to the two independent TSPs.

The DTSPMS as described here is a combination of TSP and pickup and delivery problems (PDPs), and does not seem to have been previously treated in the literature. Although being concerned with pickups and deliveries, it

differs in significant ways from the "regular" PDP as described in e.g. [3] and [4].

The main additional complication is the availability of multiple LIFO loading rows and thus the need to present a loading plan as part of the solution. One description of the regular PDP with LIFO ordering (one stack only) can be found in [2].

In the regular PDP it is necessary to make sure that each pickup is performed before the corresponding delivery. This is automatically ensured in the DTSPMS, since all pickups are performed before all deliveries.

Furthermore in the capacitated case it would suffice to check the capacity constraints for the full vehicle, since all items will need to be kept in the container at the same time.

Apart from the regular PDP, another class of problems that show similarities with the DTSPMS is the TSP with Backhauls (TSPB) (cf. e.g. [6] or [5]). Here the property that "all pickups lie before all deliveries" is preserved, however there is no longer any constraints tying a pickup to its corresponding delivery.

The DTSPMS is a special case of the generalized Pickup and Delivery Problem with Loading Constraints (2D or 3D).

The paper is organised in several parts. First a mathematical formulation of the problem is presented in Section 2 and some comments are made on its solution in GAMS/CPLEX. Next, three different heuristic solution approaches are presented in Section 3, with emphasis on the developed neighbourhood structure that is used for all approaches. Finally, Section 4 describes the implementations and gives some computational results, and Section 5 concludes on the presented solution approaches, and gives some ideas for future work on the DTSPMS.

2 Mathematical Formulation

The DTSPMS has been modelled as a binary integer programming problem with variables

$$x_{ij}^{\tau} = \begin{cases} 1, \text{ if edge } (i,j) \text{ is used in graph } \tau \\ 0, \text{ otherwise} \end{cases} , \forall i,j \in V^{\tau}$$

$$y_{ij}^{\tau} = \begin{cases} 1, \text{ if } v_i^{\tau} \text{ is visited before } v_j^{\tau} \\ 0, \text{ otherwise} \end{cases} , \forall i,j \in V_C^{\tau}$$

$$z_{ir} = \begin{cases} 1, \text{ if item } i \text{ is placed in row } r, \\ 0, \text{ otherwise} \end{cases} , \forall i \in R$$

Now let the cost of edge (i, j) in graph τ be c_{ij}^{τ}. The objective function can then be expressed as:

$$\min \sum_{\substack{i,j \in V^{\tau} \\ \tau \in \{\mathrm{P},\mathrm{D}\}}} c_{ij}^{\tau} \cdot x_{ij}^{\tau}$$

The constraints can be stated as follows:

$$\sum_{i} x_{ij}^{\tau} = 1 \qquad \forall j \in V^{\tau} \tag{1}$$

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

$$y_{ij}^{\tau} + y_{ji}^{\tau} = 1 \qquad \forall i, j, \tau, i \neq j \tag{3}$$

$$y_{ik}^{\tau} + y_{kj}^{\tau} \leq y_{ij}^{\tau} + 1 \qquad \forall i, j, k, \tau \tag{4}$$

$$x_{ij}^{\tau} \leq y_{ij}^{\tau} \qquad \forall i, j, \tau \tag{5}$$

$$y_{ij}^{\mathrm{P}} + z_{ir} + z_{jr} \leq 3 - y_{ij}^{\mathrm{D}} \qquad \forall i, j, r \tag{6}$$

$$\sum_{r} z_{ir} = 1 \qquad \forall i \tag{7}$$

$$\sum_{i} z_{ir} \leq L \qquad \forall r \tag{8}$$

$$x, y, z \in \mathbb{B} \tag{9}$$

where i, j and k are in V_C^{τ} unless otherwise stated.

Constraints (1) and (2) are flow conservation contraints, stating that one unit of flow must come in and go out of each node.

Constraints (3) ensure that for each pair of nodes i, j the precedence variable must be set, i.e. either i is before j or j is before i. (4) express that if i is before k and k is before j, then i must necessarily be before j and constraints (5) make sure that if the edge (i, j) is used, then the precedence variable is set accordingly (i is visited before j).

Constraints (6) express the LIFO constraints that are only relevant when two items in the same row, i.e. if i and j are placed in the same row, and i is picked up before j, then i must be delivered after j (i may not be delivered before j).

Finally, (7) ensure that all items must be assigned to exactly one row, and (8) enforce the row capacities L.

The model has been implemented in GAMS/CPLEX, which was able to solve problems for container sizes up to 2 by 5 or 3 by 4 within an hour of running time. Since the smallest real-life instance is of size 3 by 11 it was therefore decided to attempt to solve the problems heuristically.

3 Heuristic Solution Approaches

Since the mathematical model was unsolvable by a standard solver for realistic problem sizes, a number of metaheuristic solution approaches have been considered for this problem.

Tabu Search (TS) has previously presented good solutions to Vehicle Routing Problems, which are similar of nature to the current problem, although it seems to be less popular for the Pickup and Delivery problem.

Simulated Annealing (SA) is another method that is well-known to provide good results to many different problems. Its advantage over TS is that since it does not check the entire neighbourhood, it can move faster through a number of neighbourhoods, and thus complete a higher number of iterations in a given time.

Finally a simple *Steepest Descent* (SD) approach with restarts is considered to determine the extend to which local optima exist in the solution space.

Each of these three solution approaches solves a problem by local search, based on an initial solution and some neighbourhood structure/operator, which once developed can therefore be reused for more than one approach.

For the DTSPMS two different operators have been developed, which in combination can cover the entire solution space. These two operators have been implemented for use with each of the three above-mentioned approaches. Both operators preserve feasibility of the solution.

For comparability all three approaches use elapsed time as stopping criterion, and each has then adapted to make best use of a fixed running time.

3.1 Initial Solution

A feasible solution to the problem can be found by solving the single-stack version of the problem. In this case the pickup and delivery orderings must be exactly opposite, and the solution can then be obtained by adding the two graphs and solving a regular TSP on the resulting graph. Using a savings-algorithm to solve the TSP, this initial solution has been used for the TS and SA implementations, since these only need one initial solution to the problem.

For the steepest descent with restarts, a number of initial solutions were constructed by randomly generating an ordering of the items to use for the pickup route, and reversing this for the delivery route, thus still solving the single-stack problem. Loading rows were assigned randomly by partitioning all orders evenly among the available rows.

3.2 Neighbourhood Structures

In this section the two operators that form the basis of the solution of the DTSPMS are introduced.

Route-Swap

The first operator only performs changes to the routing of the two tours, and leaves the row assignment untouched. The neighbourhood consists of all possible swappings of two items A and B in a route where they immediately follow each other. During this operation it is necessary to consider whether the two items are placed in the same row, to ensure feasibility of the resulting solution. These two cases can be seen in Figure 1.

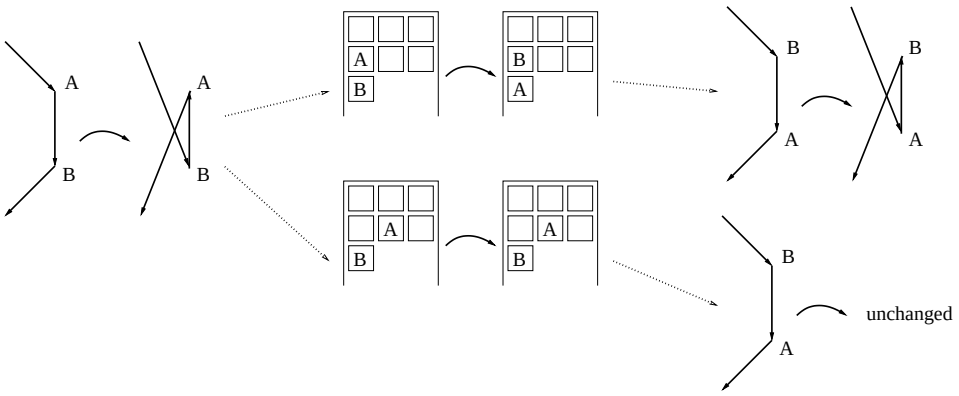

Figure 1: An illustration of *route-swap*

If A and B are placed in the same row, then they will be neighbours in this row and their positions in the loading plan of the container will be swapped by swapping their pickup order. Consequently their positions must also be swapped in the opposite route (even when A and B are not consecutive in the opposite route).

If A and B are placed in separate loading rows, then, since they are consecutive in the pickup route, this means that when v_A^P is visited, v_B^P will be next in line and since A and B are in separate rows, then the loading position of B will also be accessible (and be the last accessible position in its row), and thus the positions of A and B in the final loading plan can remain unchanged when swapping the pickup order.

This argument can be reversed when the operator is used on the delivery route.

The entire neighbourhood can be traversed by considering all values of $i = 1, \ldots, n-1$ for both routes τ, thus the size of this neighbourhood is $2n-2$,

i.e. $O(n)$, where n is the number of orders.

Complete-Swap

The operator *complete-swap* is focusing on the row assignment, while only updating the routes to maintain feasibility. It considers a pair of items that are currently assigned to different rows, and swaps their positions in the row assignment, and in each of the tours. This is illustrated in Figure 2.

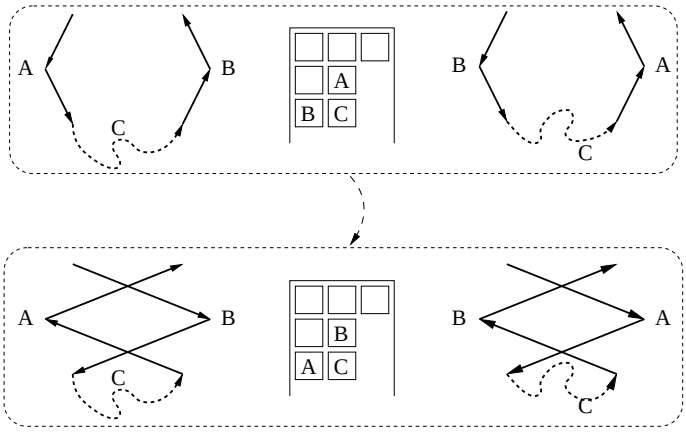

Figure 2: An illustration of *complete-swap*

To traverse the entire neighbourhood one must examine all pairs of orders (skipping pairs where both orders are assigned to the same row), and the size of the neighbourhood is therefore $O(n^2)$.

The operators in combination

Since *route-swap* does not affect the row assignment of a solution, it is obvious that one cannot reach the entire solution space by only using this operator. Similarly when performing *complete-swap* the mutual visiting orders of the two routes will never change (e.g. if a customers is visited third in the pickup route and last in the delivery route, then there can never be a customer visited third in the pickup route and not last in the delivery route, if only *complete-swap* is applied). However by using a combination of the two operators it becomes possible to cover the entire solution space.

3.3 Tabu Search

In the tabu search implementation the fact that two different neighbourhood structures must be considered has been dealt with by systematically chang-

ing between the two types, following some pre-defined pattern. Two parameters are used to describe this pattern, namely the *length* or *period* of the pattern, `perLen`, and the *ratio* between the two operators, `ratio`. E.g. the parameter combination $\texttt{perLen} = 20, \texttt{ratio} = 0.3$ indicates that the first 6 iterations should use *route-swap*, the next 14 should use *complete-swap*, and from iteration 21 this pattern repeats itself.

In this way the deterministic nature of tabu search has been maintained, by not introducing randomness in the selection of the operator to use for each iterations.

For each operator it has additionally been necessary to decide on some attributes to register in the tabu list, and for both operators the choice has been to register the two orders that were swapped.

For moves of type *route-swap* it was thus *not* registered which route (pickup or delivery) the swap was performed for. When swapping two visits that are in the same row the swap affects both routes, and thereby justifies this. When swapping two visits assigned to different rows, swapping the same two customers on the opposite route would not imply a reversion of the previous move, but for the sake of simplicity this was found to be acceptable.

Although moves of both types were marked by the same attributes in the tabu list, a separate list was kept for each move type. This is due to the differences in the two operators. Consecutively performing one move of each type on the same pair of orders does not imply that one is reversing the other, and does not lead the algorithm to start cycling between a few solutions.

The TS algorithm implemented here is rather simple, with a constant tabu list length `tabuLength`, and a stopping criterion based on elapsed time/number of iterations. To ensure comparability with the other approaches it was decided not to stop after a given number of iterations without improvement, but simply run for the entire time span allotted.

3.4 Simulated Annealing

The SA implementation handles the two available operators by randomly selecting one of them for each iteration, and the probability for choosing each operator is expressed in the parameter `ratio`, indicating the probability of choosing *route-swap*.

Traditionally SA approaches take three parameters; an initial temperature T_s, a final temperature T_e, and a temperature reduction function. Additionally one must decide if the temperature should be updated for each iteration, or at regular intervals. For simplicity the temperature has been updated at each iteration in the implementation described here.

The temperature reduction function is often based on the reduction scheme

$$T_{i+1} = c \cdot T_i \tag{10}$$

where $c \in (0; 1)$ is some reduction factor, and the algorithm terminates when the final temperature T_e is reached.

However for the purpose of comparison SA has here been implemented to take running time as an input parameter instead of the temperature reduction factor.

Adjustment of the cooling scheme

The temperature reduction function given in (10) is equivalent to finding the temperature at iteration x, $T(x)$, using the formula:

$$T(x) = T_s \cdot c^x \tag{11}$$

where T_s is the starting temperature, c is the reduction factor and x is the iteration counter, and the algorithm is stopped once the desired final temperature T_e is reached.

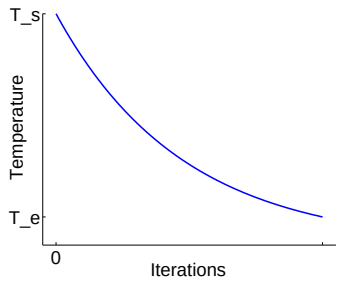

Figure 3: Standard cooling scheme

For the case at hand this has been modified to allow the algorithm a predetermined running time, and still ensure that an appropriate temperature interval is covered, effectively keeping the same temperature reduction curve with changed x-axis.

The cooling scheme is constructed such that the desired initial and final temperatures T_s and T_e are returned at times $t = 0$ and $t = t_{\max}$ respectively, and the temperature decreases exponentially between these points.

This means that the iteration count x is replaced by the percentage of elapsed time, $t \in [0; 1]$, and the reduction factor c must then be selected such that $T(0) = T_s$ and $T(1) = T_e$, i.e. $c = \frac{T_e}{T_s}$. Insertion into the initial expression then gives

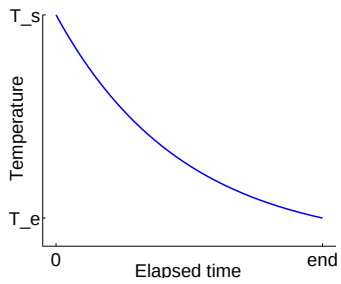

Figure 4: Adjusted cooling scheme

$$T(t) = T_s \cdot \left(\frac{T_e}{T_s}\right)^{\frac{t}{t_{\max}}}$$

This expression can easily be modified to work on the maximum number of iterations instead of the total running time.

4 Computational Results

Each of the three algorithms has been implemented in Java and tests have been performed for running times 10 seconds and also for running times 3 minutes (both wall clock times). The choice of using wall clock times is due to the fact that it is difficult to measure CPU-times in Java, and that the company that introduced the problem were interested in the running times that would be experienced by the customers. The 10 second interval was chosen to ressemble online computations, while 3 minutes was expected to produce considerably better solutions within a duration that users would still find worth the wait.

Unsurprisingly, wall clock time showed to be an unreliable measure of performance, since the algorithms have proven somewhat sensitive to randomness and two "identical" runs could produce considerable variation in solution quality. Therefore each algorithm has initially been run three times on an otherwise idle computer, with all possible combinations of the sensitive parameters (i.e. the parameters that affect the average execution time of each iteration), to measure the number of iterations corresponding to three minutes running time. The number of iterations used for the 10 second running times have been found by scaling.

All timed tests have been performed on a Dell D610 with 1.5 GB RAM and a 1.60 GHz processor.

4.1 Test Instances

The test instances that have been used for the evaluation of the solution approaches have been generated randomly, by finding two sets of n random points in a 100×100 square. The depot is placed in the centre of the square at $(50, 50)$. All distances are Euclidean distances rounded to the nearest integer, in accordance with the conventions from TSPLIB.

Two test sets have been generated, each containing ten problems of size $n = 33$, i.e. in total 20 problems have been generated. The first set was used for parameter tuning for each of the heuristics, and the second set was used for the final results reported in the final Table 9.

The test instances can be obtained from http://www.imm.dtu.dk/∼hlp.

In this paper all test problems have been solved for the 3-stack problem.

4.2 Bounds

Since the optimal solution to the test instances is unknown, it is desirable to find good lower bounds to evaluate the quality of the solutions produced by the test runs.

One lower bound can be found by relaxing the constraints that ensure loading feasibility between the graphs, i.e. solving the n-stack problem. This way the problem reduces to two individual TSPs, that can be solved to optimality with tools such as Concorde (see [1]), and the sum of the lengths of these two tours is then a lower bound on the sum of the lengths of the tours in any feasible solution to the DTSPMS with any number of rows.

However, intuitively this bound must be expected to be quite weak, since one would expect many changes to be necessary before a feasible loading plan could be constructed for such a solution.

The best known solution to each of the test problems has been compared with this lower bound in Table 9 at the end of this section.

4.3 Results

A number of test runs have been performed for each of the three algorithms implemented. These will first be presented individually before presenting the final results and comparisons in Table 9.

All three algorithms required considerations regarding the ratio between the two different operators, and TS and SA each include a number of parameters that additionally need to be estimated.

All calibration runs were performed on the problems from the first data set to determine a good set of parameters, and these parameters were then applied to the problems from the second data set to produce the final results presented in Table 9 on page 17.

Wherever possible the tests have been repeated three times with different seeding of the random number generator, and the presented results are averages over these three runs. This technique has been used throughout for steepest descent and simulated annealing, but was not possible for tabu search, which is completely deterministic in the implementation used here.

Whenever *solution quality* is mentioned in the following, this refers to "deviation from the best known solution", i.e. the objective value of the solution to a given problem is divided by the value of the best known solution to that problem, and then this value is averaged over all problems considered.

Steepest Descent

For each iteration of the steepest descent algorithm the operator to use was chosen randomly with some probability *ratio*, similarly to the way it is done for simulated annealing.

Since the entire neighbourhood is searched for each iteration, and the two neighbourhoods have different size, the number of iterations completed in a given time span must therefore be expected to vary with the ratio.

The algorithm was first run three times to determine the number of iterations that could on average be completed in three minutes with different ratios. The results of these tests are shown in Table 1.

0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
2124	2210	2352	2625	3147	4044	5534	7845	11615

Table 1: Number of completed runs for Steepest Descent in three minutes.

Using the numbers from Table 1, different values of *ratio* were examined and compared with regard to solution quality, as reported in Table 2.

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
10 s.	1.574	1.568	1.566	**1.563**	1.567	1.568	1.576	1.582	1.582
3 m.	1.550	1.534	1.521	**1.518**	1.529	1.532	1.558	1.575	1.581

Table 2: Solution quality depending on *ratio* for Steepest Descent.

Based on Table 2 a ratio of 0.4 was used for the final tests for both of the running times considered.

Tabu Search

For the tabu search implementation the number of iterations that could be completed within three minutes could be expected to vary with (1) the *ratio* between the two operators, since their neighbourhoods are of different size, and possibly (2) the length of the tabu list *tabuLength*, since this determines the number of solutions that must be checked for tabuness in each iteration. However, in comparison (2) showed to have very little influence and was therefore removed. Table 3 shows the average number of iterations completed for different values of *ratio*, with $tabuLength = 9, perLen = 10$.

0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
54	60	68	77	88	108	138	185	282

Table 3: Number of iterations (in thousands) completed in 3 minutes for TS.

Next the length of the tabu list, *tabuLength*, was examined. The results are shown in Table 4, and have been found with $ratio = 0.5, perLen = 10$.

	5	7	9	11	13
10 s.	1.363	**1.334**	1.341	1.350	1.350
3 min.	1.318	1.257	1.249	**1.233**	1.249

Table 4: Solution quality depending on length of the tabu list.

Performing some manual, random checks for cycling shows that cycles occur regularly for $tabuLength = 5$, but none have been discovered for higher values of *tabuLength*.

Based on these results tabu lengths of 7 and 11 were used for the remaining runs, for 10 seconds and 3 minutes respectively.

Finally the combination of the operators was examined, given by the parameters *ratio* and *perLen* as described in Section 3.3. The results of these runs can be found in Table 5.

Based on Table 5, parameters $ratio = 0.9, perLen = 10$ for 10 seconds and $ratio = 0.7, perLen = 10$ for 3 minutes were used for the final tests.

Simulated Annealing

For the SA-algorithm some test runs were first performed to determine the potential temperature interval. During these runs all discoveries of a new best solution were recorded, and Figure 5 shows the best seen solution at any time as a function of the temperature for the ten different test problems from set 0.

10 sec.

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
10	1.394	1.368	1.364	1.363	1.334	1.339	1.342	1.342	**1.339**
20	1.386	1.367	1.358	1.354	1.360	1.348	1.343	1.354	1.335
50	1.388	1.380	1.369	1.390	1.370	1.355	1.357	1.362	1.349
100	1.387	1.408	1.384	1.391	1.383	1.378	1.364	1.372	1.362

3 min.

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
10	1.244	1.250	1.243	1.233	1.233	1.228	**1.213**	1.234	1.228
20	1.266	1.256	1.250	1.230	1.236	1.218	1.231	1.214	1.207
50	1.269	1.253	1.256	1.252	1.237	1.256	1.227	1.229	1.227
100	1.282	1.274	1.267	1.263	1.262	1.255	1.253	1.247	1.225

Table 5: Solution quality depending on *ratio* and *perLen*.

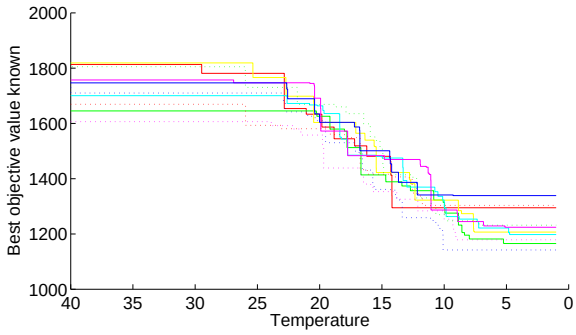

Figure 5: Indication of temperature intervals with potential.

As can be seen from the figure it makes little sense to use starting temperatures above 25, and at temperatures lower than 5 there are only few global improvements. The plot has been made for a running time of 3 minutes, but the numbers are similar for both longer and shorter running times, although improvements occur in a slightly larger interval with longer running times. Accordingly it was decided to closer examine initial temperatures $T_s \in \{20, 22, 24, 26\}$ and final temperatures $T_e \in \{3, 5, 7, 9, 11\}$. For each combination of these temperatures three runs were performed, and the average number of iterations, as shown in Table 6, was then used for the actual test runs. Note that different scales are used in Tables 3 and 6.

The number of iterations completed in a given timespan increases with lower temperature (since more solutions are rejected at these temperatures), which made it necessary to determine the number of iterations for each temperature interval.

As SA selects a random neighbour and does not need to examine the entire neighbourhood the ratio between the move types should not affect the number of iterations that can be completed in a given time interval. Thus it is fair to compare different ratios for an identical number of iterations.

	20	22	24	26
3	14.9	14.8	14.8	14.7
5	14.7	14.6	14.5	14.5
7	14.7	14.4	14.3	14.3
9	14.3	14.2	14.2	14.1
11	14.2	14.1	14.0	13.9

Table 6: Number of iterations (in millions) completed in 3 minutes for SA.

Next the importance of the *ratio* was examined, performing a series of runs with different values. These runs all used the same temperature values from the centre of the interval $T_s = 22, T_e = 7$, and the results are reported in Table 7.

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
10 s.	1.21	1.20	1.20	**1.18**	1.19	1.19	1.20	1.21	1.22
3 m.	1.12	1.11	1.11	1.10	**1.09**	1.10	1.11	1.11	1.12

Table 7: *ratio* - Percentage of iterations using *route-swap* for SA.

As can be seen from the table the choice of ratio has surprisingly little impact on the achievable solution quality. Based on these runs a ratio of 0.4 was selected for use through the remaining experiments for 10 second runs, and 0.5 for the remaining 3 minute runs.

The same runs were performed with ratios 0.0 and 1.0, to demonstrate the

effect of using only one operator. As expected this approach produced worse solutions, and the solution quality was 1.47 (*route-swap*) and 1.35 (*complete-swap*) when using one operator exclusively.

Finally all combinations of start and end temperatures was examined, and Table 8 shows the results.

10 sec.	20	22	24	26
3	1.202	1.190	1.191	1.208
5	1.183	**1.189**	1.192	1.182
7	1.195	1.177	1.188	1.203
9	1.211	1.221	1.217	1.222
11	1.253	1.261	1.251	1.270
3 min.	20	22	24	26
3	1.106	1.098	1.098	1.095
5	**1.090**	1.103	1.100	1.094
7	1.109	1.105	1.103	1.105
9	1.117	1.122	1.124	1.124
11	1.151	1.162	1.169	1.162

Table 8: Solution quality obtained for different temperature intervals.

Based on Table 8 it was decided to use the temperature settings $T_s = 20, T_e = 5$ for three minute runs, and $T_s = 22, T_e = 5$ for 10 second runs.

Summary of the Results

After determining a good set of parameters for each of the three heuristics, Table 9 summarizes the results obtained by applying these parameters to all test problems.

The first column gives the problem number. Problems R00-R09 constitute the first data set, which was used for calibration, while problems R10-R19 constitute the second set and have only been used for the final evaluation. The second column (best) shows the value of the best known solution to each problem, nS gives a lower bound on the optimal solution (the optimal solution to the n-stack problem), SS gives the optimal solution to the single-stack problem, while the column *init* gives the initial solution that was used for TS and SA.

The remaining columns demonstrate the quality of the solutions that have been found with each of the three heuristic approaches for the two different running times. All values are found using the "best" set of parameters as described in the previous sections, and dividing the result obtained for each problem by the best known solution for that problem. For SA and SD three

					10 seconds			3 minutes		
	Best	nS	SS	init	SA	TS	SD	SA	TS	SD
R00	1069	914	1886	1685	1.25	1.41	1.65	1.12	1.23	1.57
R01	1072	875	1690	1581	1.13	1.29	1.58	1.04	1.16	1.55
R02	1070	935	1672	1563	1.18	1.37	1.56	1.10	1.23	1.52
R03	1111	961	1836	1745	1.21	1.42	1.65	1.11	1.23	1.61
R04	1090	933	1671	1628	1.21	1.33	1.53	1.07	1.19	1.50
R05	1055	898	1548	1439	1.17	1.20	1.46	1.10	1.16	1.41
R06	1118	998	1739	1644	1.18	1.36	1.55	1.10	1.25	1.50
R07	1118	962	1867	1695	1.21	1.37	1.64	1.09	1.25	1.56
R08	1111	976	1761	1636	1.21	1.35	1.56	1.11	1.26	1.51
R09	1106	982	1610	1553	1.13	1.28	1.46	1.06	1.17	1.44
R10	1021	901	1697	1575	1.23	1.44	1.66	1.14	1.25	1.63
R11	1040	892	1494	1429	1.19	1.20	1.44	1.06	1.18	1.42
R12	1113	984	1778	1673	1.21	1.43	1.60	1.12	1.22	1.55
R13	1102	956	1707	1613	1.21	1.35	1.53	1.06	1.20	1.50
R14	1059	879	1704	1565	1.19	1.44	1.59	1.09	1.22	1.55
R15	1162	985	1943	1783	1.19	1.35	1.60	1.08	1.24	1.54
R16	1105	967	1767	1647	1.19	1.34	1.59	1.09	1.17	1.54
R17	1096	946	1716	1620	1.21	1.45	1.56	1.10	1.25	1.55
R18	1180	1008	1796	1673	1.15	1.26	1.52	1.08	1.16	1.46
R19	1123	938	1725	1633	1.14	1.28	1.54	1.08	1.22	1.50
Avg. set 0					1.19	1.34	1.56	1.09	1.21	1.52
Avg. set 1					1.19	1.35	1.56	1.09	1.21	1.52

Table 9: Result Summary

runs have been performed with different random seeds, and the reported values are averages over these runs.

The best known solutions have all been found with the SA algorithm running times of 15-20 minutes. Using running times longer than this does not seem to produce further improvement. Additionally, the temperature interval leading to the best known solution varies over most of the intervals that were considered for the tests reported in Table 8, with an overweigth around $T_s = 22$ and $T_e = 5$.

Comparisons show that the best known solutions are all within 12-23% of the lower bound (nS). Given that the lower bound is found by completely ignoring all sequencing constraints, one may assume that these lower bounds are quite weak. This suggests that the best found solutions are reasonably close to optimal.

The initial solutions (init) are all found by applying a savings algorithm to the single-stack problem. These values can be compared to the optimal solution to the SS problem to confirm that the choice of the savings algorithm does not considerably reduce the quality of the initial solutions. Thus an attempt to construct better initial solutions should focus on improving the structure of the solution (look beyond the single-stack problem), rather than looking for better solutions to the SS problem.

The top half of the table shows the objective values obtained for the data set that was used for parameter tuning, while the bottom half shows the results for a "fresh" data set, and averages for each set are reported at the bottom of the table. This shows that the solution quality is not significantly higher for the problems that have been used for calibration, i.e. the choice of calibration problems does not seem to have influenced the final parameter values.

Table 9 shows that of the three implementations SA consequently produces better results than the two other algorithms, and gives results around 20% above the best known for a running time of 10 seconds, and around 10% when allowed to run for 3 minutes.

For both tabu search and simulated annealing a significant improvement can be observed by increasing the running time, while only a small improvement can be obtained by increasing the running time for steepest descent with restarts.

The results also show that using steepest descent with the short running time produces solutions that are only slightly better than the initial solutions.

5 Conclusion and future work

This paper has introduced a new variant of the TSP/PDP, presented a mathematical formulation of it, and shown some ways to solve it heuristically.

Three metaheuristic solution approaches have been implemented, and comparisons show that Simulated Annealing seems to produce the best result of the three, with solutions that are on average within 10-12% of the best known solution, with a running time of 3 minutes, and 20-25% for running times of 10 seconds.

5.1 Future work

Future work on this problem could focus on either improving the solutions/solution methods or generalising the problem.

One way of improving the solutions, would be by improving the approaches described here. In particular one could attempt refining the tabu search, e.g. by using variable tabu list length, or by introducing some diversification mechanism, such as performing mutations to the solution when a certain number of iterations have been performed without leading to any improving solutions.

Additionally it could be attempted to solve the DTSPMS to optimality, and it could also be interesting to examine the effect of the number of loading rows on the solution to the problem.

The most obvious extension of the problem is to either generalise the TSP aspects of the problem and include multiple vehicles/containers and/or multiple depots to form a DVRPMS, or to generalise the problem in the direction of the regular pickup and delivery problem, to form a PDPMS.

In the first case, if the choice of depot becomes a decision, it is likely that the cost of the long-haul between the depots becomes a factor, and this may be very hard to estimate at the time of planning.

The consideration of time windows in connection with this problem, will most likely not be relevant. Either these will be very wide, and thus have a very small impact on the final solution, or they will be very narrow, and in that case will have a very big impact on the solution, either rendering the problem infeasible if the time windows collide with the precedence constraints, or the time windows will simply become controlling of the final solution and thereby making the precedence constraints superfluous.

Modifying the problem to include non-uniform objects would complicate the packing considerably, since some objects might still be positioned next to each other, so that their internal delivery order becomes irrelevant. This

would heavily influence the use of the LIFO-principle so far, and would approach the problem to the more generalised PDP with loading constraints.

References

[1] David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. Implementing the Dantzig-Fulkerson-Johnson algorithm for large traveling salesman problems. *Mathematical Programming*, 97(1-2):91–153, 2003.

[2] Francesco Carrabs, Jean-François Cordeau, and Gilbert Laporte. Variable Neighborhood Search for the Pickup and Delivery Traveling Salesman Problem with LIFO Loading. Technical report, C.R.T. Report, 2005/12.

[3] Jean-François Cordeau, Gilbert Laporte, Jean-Yves Potvin, and Martin W.P. Savelsbergh. Transportation on Demand. *Transportation*, 2006.

[4] Guy Desaulniers, Jacques Desrosiers, Andreas Erdmann, Marius M. Solomon, and François Soumis. VRP with Pickup and Delivery. In Paolo Toth and Daniele Vigo, editors, *The Vehicle Routing Problem*, chapter 9, pages 225–242. SIAM, 2001.

[5] Michel Gendreau, Alain Hertz, and Gilbert Laporte. The traveling salesman problem with backhauls. *Computers & Operations Research*, 23(5):501–508, 1996.

[6] Paolo Toth and Daniele Vigo. A heuristic algorithm for the symmetric and asymmetric vehicle routing problems with backhauls. *European Journal of Operational Research*, 113(3):528–543, 1999.