A Branch-and-Cut Algorithm for the Double Traveling Salesman Problem with Multiple Stacks

Manuel Alba* Jean-François Cordeau† Mauro Dell'Amico*
Manuel Iori*

December 7, 2010

Abstract

The double traveling salesman problem with multiple stacks is a variant of the pickup and delivery traveling salesman problem in which all pickups must be completed before any of the deliveries. In addition, items can be loaded on multiple stacks in the vehicle and each stack must obey the last-in-first-out policy. The problem consists in finding the shortest Hamiltonian cycles covering all pickup and delivery locations while ensuring the feasibility of the loading plan. We formulate the problem as two traveling salesman problems linked by infeasible path constraints. We also introduce several strengthenings of these constraints which are used within a branch-and-cut algorithm. Computational results performed on instances from the literature show that the algorithm outperforms existing exact algorithms. Instances with up to 28 requests (58 nodes) have been solved to optimality.

Keywords: Traveling salesman problem, pickup and delivery, LIFO loading, branch-and-cut.

1 Introduction

In the classical *Pickup and Delivery Traveling Salesman Problem* (PDTSP), a single uncapacitated vehicle must serve a set of transportation requests, each defined by an origin-destination pair. The problem consists in finding a least-cost Hamiltonian cycle

*DISMI, University of Modena and Reggio Emilia, Via Amendola 2, Padiglione Buccola, 42122, Reggio Emilia, Italy.

†Canada Research Chair in Logistics and Transportation and CIRRELT, HEC Montréal, 3000 chemin de la Côte-Sainte-Catherine, Montréal, H3T 2A7 Canada.

on the pickup and delivery vertices with the additional constraint that the delivery vertex of any given request must be visited after the corresponding pickup vertex. The *Double Traveling Salesman Problem with Multiple Stacks* (DTSPMS) is a variant of the PDTSP in which all pickups must be performed before all deliveries, and collected items can be inserted in multiple stacks in the vehicle. The vehicle must first perform a Hamiltonian cycle on the set of pickup vertices before it performs a second Hamiltonian cycle on the set of delivery vertices. Each customer request consists of one item and the vehicle has a loading space divided into stacks of fixed height which must obey the *Last-In-First-Out* (LIFO) policy: each loaded item is to be placed at the top of a stack and only the items located at the top of a stack can be unloaded from the vehicle. A routing cost is associated with the arcs between the vertices, and the aim of the problem is to find two Hamiltonian cycles (one on the pickup vertices and one on the delivery vertices) of minimum total cost for which there exists a feasible loading plan satisfying the stack heights and the LIFO policy.

A simple example with four customers and a vehicle having $s = 2$ stacks of height $l = 2$ is represented in Figure 1, where, from left to right, the pickup tour, the delivery tour and a feasible loading plan are depicted. The pickup tour starts from and ends at the pickup depot, located at vertex 0 of the pickup region. While visiting the pickup points, the items are loaded into the stacks from bottom to top. The vehicle then moves to the delivery region. The second tour visits all delivery points and unloads the corresponding items from the top of the stacks.

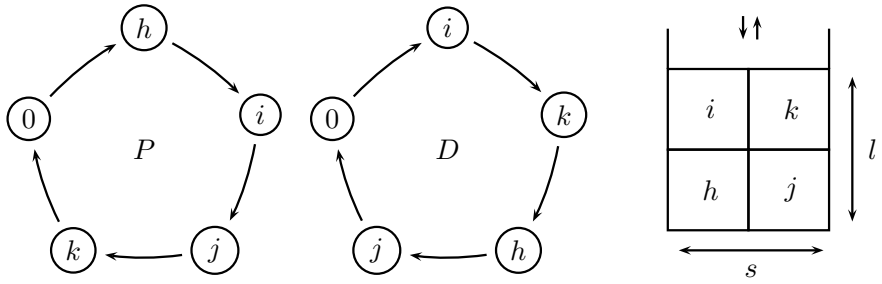

Figure 1: A simple DTSPMS example, with a pickup tour (left), a delivery tour (middle) and a loading plan (right).

The DTSPMS was introduced by Petersen and Madsen (2009) and was motivated by a real-world application arising in the transportation of pallets by trucks between a pickup and a delivery region. It belongs to the class of combined routing and loading problems: the routing part consists in solving a *Traveling Salesman Problem* (TSP) for each region, while the loading part consists in finding a feasible loading plan for the items.

Petersen and Madsen (2009) first proposed a mathematical formulation of the DTSPMS and a simulated annealing heuristic that was computationally tested on a set of instances which are now widely used as benchmarks. Later, Lusby et al. (2010) proposed an exact algorithm based on two iterative phases: in the first phase, the k best solutions are generated for each of the two separate TSPs; in the second phase, one attempts to find a feasible loading for given couples of TSP solutions. The process is iterated, considering solution pairs in non-decreasing order of total cost, until a feasible loading is found. Petersen et al. (2010) also studied several mathematical models and proposed branch-and-cut algorithms. One of these approaches, based on a two-index vehicle flow formulation with additional infeasible path constraints generated dynamically, clearly obtained better results than the others. Our work is based on a similar approach. Very recently, Carrabs et al. (2010) presented an enumerative branch-and-bound algorithm for the special case in which the vehicle has exactly two stacks.

With respect to heuristic solution approaches, Felipe et al. (2009a,b) presented neighborhood structures and a variable neighborhood search algorithm, whereas Côté et al. (2009) introduced a large neighborhood search procedure. The complexity of the DTSPMS and of several subproblems that may arise from it is discussed by Toulouse and Wolfler Calvo (2009), Casazza et al. (2009), and Bonomo et al. (2010). Finally, an approximation scheme was proposed by Toulouse (2010).

The DTSPMS is also related to the *PDTSP with LIFO Loading* (PDTSPL) which can be seen as a variant of the DTSPMS in which the vehicle has a single LIFO stack, and pickups do not have to be completed before deliveries can start. Several algorithms were proposed for the PDTSPL: variable neighborhood search (Carrabs et al., 2007b), additive branch-and-bound (Carrabs et al., 2007a) and branch-and-cut (Cordeau et al., 2010). For surveys on pickup and delivery problems and on combined routing and loading, we refer the reader to Cordeau et al. (2007) and to Iori and Martello (2010), respectively.

The contribution of this paper is to introduce valid inequalities and a new branch-and-cut algorithm for the DTSPMS. Computational experiments show that this new algorithm outperforms existing exact algorithms and can solve some instances with up to 28 requests (58 nodes).

The rest of this paper is organized as follows. The next section provides a formal definition and a mathematical formulation of the problem. Section 3 focuses on solving the loading problem. This is followed by valid inequalities in Section 4 and by the branch-and-cut algorithm in Section 5. Computational experiments are then reported in Section 6 and are followed by the conclusion.

2 Definition and Mathematical Formulation

To model the DTSPMS we start from the classical two-index vehicle flow formulation for the TSP, with the addition of infeasible path constraints to represent the loading subproblem. This formulation is based on the idea of decomposing the problem into its routing and loading components. The goal of the routing component is to construct two optimal TSP tours, one for the pickup region and one for the delivery region. Then, by iteratively solving the loading subproblem, we identify cuts that eliminate (partial) tours that do not allow the construction of a feasible loading plan.

More formally, let n denote the number of customer requests. We define the DTSPMS on two complete directed graphs $G^P = (V^P, A^P)$ and $G^D = (V^D, A^D)$, where V^P and A^P are, respectively, the vertex and arc set in the pickup region, and V^D and A^D the vertex and arc set in the delivery region. We make use of the notation $G^T = (V^T, A^T)$, with $T \in \{P, D\}$, to define properties that apply to both graphs.

For $T \in \{P, D\}$, we define $V^T = \{0^T\} \cup V_0^T$, where vertex 0^T represents the depot at which each tour starts and ends. Subsets $V_0^P = \{1^P, \ldots, n^P\}$ and $V_0^D = \{1^D, \ldots, n^D\}$ represent the sets of pickup and delivery vertices, respectively. Each request i is associated with pickup vertex i^P and delivery vertex i^D, $i = 1, \ldots, n$. When no confusion arises we will use the symbol i to denote both i^P and i^D. Each arc $(i, j)^T \in A^T$ has a non-negative routing cost c_{ij}^T, $T \in \{P, D\}$. Without loss of generality we suppose that the routing cost from the pickup depot 0^P to the delivery depot 0^D is zero.

The demand of each customer request i consists of a single unit-size item (e.g., a pallet), that has to be loaded when visiting i^P and unloaded at i^D. The vehicle has a loading space composed of s stacks, each of which can accommodate at most l items. The DTSPMS requires that the LIFO policy be satisfied: if the pickup vertex i^P is visited before the pickup vertex j^P and i^P and j^P are loaded into the same stack, then the delivery vertex j^D must be visited before the delivery vertex i^D.

The DTSPMS consists in finding two tours, one for each region, of minimum total cost, starting from depot 0^T, visiting every vertex in V_0^T exactly once, and ending at depot 0^T, for $T \in \{P, D\}$.

To formulate the DTSPMS, we associate to each arc $(i, j)^T \in A^T, T \in \{P, D\}$, a binary variable x_{ij}^T taking value 1 if and only if vertex j^T is visited immediately after vertex i^T by the vehicle. The existence of the two tours, independently from the loading problem, can then be formulated as the following integer linear program.

$$\text{Minimize} \quad \sum_{\substack{(i,j)^T \in A^T \\ T \in \{P,D\}}} c_{ij}^T \; x_{ij}^T \tag{1}$$

subject to

$$\sum_{j \in V^T} x_{ij}^T = 1 \qquad i \in V^T, T \in \{P, D\} \tag{2}$$

$$\sum_{i \in V^T} x_{ij}^T = 1 \qquad j \in V^T, T \in \{P, D\} \tag{3}$$

$$\sum_{i \in S} \sum_{j \in S} x_{ij}^T \leq |S| - 1 \qquad S \subseteq V^T,\ |S| \geq 2,\ T \in \{P, D\} \tag{4}$$

$$x_{ij}^T \in \{0, 1\} \qquad (i, j) \in A^T,\ T \in \{P, D\}. \tag{5}$$

The objective function (1) minimizes the total routing cost. Constraints (2) and (3) ensure that each pickup and delivery vertex is visited exactly once. Constraints (4) ensure the connectivity of the solution.

To model the loading component of the problem, we introduce an additional set of infeasible path constraints. Let us denote by $Q = \{p_1, p_2, \ldots, p_q\} \subseteq V_0^P$ a path visiting q vertices in the pickup region, and by $A(Q)$ the set of arcs used by path Q, i.e., $A(Q) = \{(p_1, p_2)^P, (p_2, p_3)^P, \ldots, (p_{q-1}, p_q)^P\}$. Similarly, let us denote by $F = \{d_1, d_2, \ldots, d_f\} \subseteq V_0^D$ a path visiting f vertices in the delivery region, and by $A(F)$ the set of arcs used by path F, i.e., $A(F) = \{(d_1, d_2)^D, (d_2, d_3)^D, \ldots, (d_{f-1}, d_f)^D\}$. We say that a pair (Q, F) of paths $Q \subseteq V^P$ and $F \subseteq V^D$ is *load-infeasible* if there exists no feasible loading of the requests belonging to both paths.

We impose the existence of a feasible loading to the associated routing solution by using the following constraint for any load-infeasible pair of paths (Q, F):

$$\sum_{j=1}^{q-1} x_{p_j, p_{j+1}}^P + \sum_{j=1}^{f-1} x_{d_j, d_{j+1}}^D \leq |A(Q)| + |A(F)| - 1. \tag{6}$$

A simple example of constraint (6) is given in Figure 2, where the first path denotes Q (i.e., the path in the pickup region) and the second F (i.e., the path in the delivery region). Let us suppose that the vehicle has two stacks. Indeed, i cannot be loaded in the same stack as j or k because of the LIFO requirement. Similarly, j cannot be loaded in the same stack as k. It follows that at least three stacks are needed to feasibly load the requests belonging to these two paths. Hence, at most nine arcs can be chosen among the ten depicted in the figure.

Determining if a pair of paths is load-infeasible can be done through specialized algorithms which will be described in Section 3.

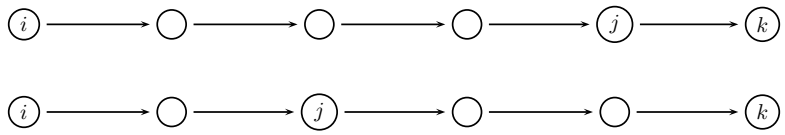

Figure 2: Graphical representation of an infeasible path constraint.

3 Solution of the Loading Problem

If no limit is imposed on the height of the stacks, then checking whether the items can be loaded into the stacks, given the pickup and delivery routes, is easy and can be done in polynomial time (see, e.g., Casazza et al., 2009). However, when the stacks are capacitated the loading problem may become difficult.

Following Bonomo et al. (2010), we define the problem of determining a feasible loading plan, if any, given two pickup and delivery tours as the *Pickup and Delivery Tour Combination* (PDTC). As it has been observed by Toulouse and Wolfler Calvo (2009), Casazza et al. (2009) and Bonomo et al. (2010), the PDTC is directly related to the *Bounded Vertex Coloring Problem* (BVCP). The BVCP generalizes the classical minimum coloring problem by imposing an upper bound on the number of times each color can be used. The PDTC can be seen as a BVCP where the number of colors corresponds to the number of stacks, and the upper bound corresponds to the fixed stack height. The BVCP is NP-hard in the general case and for any fixed value of $l \geq 3$ (Hansen et al., 1993).

In the case of the PDTC, however, the underlying graph is a *permutation graph*. On the basis of this observation, Bonomo et al. (2010) showed that the PDTC is NP-complete even if l is fixed, but can be solved in $O(n^{s^2+s+1}s^3)$ time. Hence, it is polynomially solvable for fixed s.

We solve the PDTC by using simple lower and upper bounds, followed by a complete enumerative algorithm when necessary. Let us consider two paths, a pickup path Q and a delivery path F, and let us define the subset I of customer requests appearing in both paths. To determine whether a feasible loading for the requests in I exists, we use the following procedure. We start by performing a simple check: if I contains s customers or less, then a feasible loading surely exists, as each item can be loaded alone in a different stack. Otherwise, we construct a *precedence graph* $G' = (I', A')$, where I' is composed of the elements in I plus two additional vertices: an origin vertex o and a destination vertex t, and an arc $(i, j) \in A'$ is defined if i precedes j both in the pickup path and in the delivery path, i.e., i and j cannot be loaded in the same stack. For example, the precedence graph associated to Figure 1 is given in Figure 3. Note that the original pickup and delivery depots are not included

in I' and that the precedence graph constructed in this way is acyclic.

A lower bound on the minimum number of stacks needed to load all the items can be obtained by relaxing the constraint on the maximum height of each stack and then computing a maximum clique among the vertices in I. Let us denote this clique by C. If the size of C is larger than s, then the loading is clearly infeasible. Finding the maximum clique of a general graph is NP-hard. Fortunately, the precedence graph G' is not only acyclic but also transitive: if i precedes j and j precedes k, then i precedes k (see also Figure 2). As a result, the maximum clique can be found by computing the longest path from o to t, through the classical critical path method (CPM) labeling algorithm (Kelley, 1961).

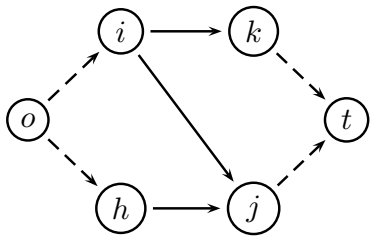

Figure 3: The precedence graph associated to the solution depicted in Figure 1.

If $|C| \leq s$ we try to compute an upper bound on the minimum number of stacks required by executing a simple greedy procedure that tries to construct a solution using the output of the CPM algorithm. This output provides, for each vertex i, a minimum index $T_{\min}(i)$ and a maximum index $T_{\max}(i)$ of the stack in which the corresponding item can be loaded. Each item i having $T_{\min}(i) = T_{\max}(i)$ (i.e., belonging to a longest path) is loaded in the corresponding stack. If by doing so we exceed the maximum height of a stack the procedure stops. Otherwise, it continues by taking into consideration the items not belonging to a longest path. Each item i is selected in non-decreasing value of $T_{\min}(i)$, breaking ties by increasing values of i, and assigned to the stack of lowest index that can accommodate it. If all items are assigned, then the feasibility of the loading problem has been proved. Otherwise, an enumeration tree is constructed.

The enumeration tree simply considers each item i in turn, in the same order as in the greedy procedure. It creates a node for any possible assignment of i to one of the stacks in the interval $[1, s]$. The tree then erases all nodes for which the maximum height of the stack or the LIFO requirements are violated, and iterates with the next item. The complexity of the enumeration procedure is $s^{|I|}$ in the worst case, but the procedure turns out to be very fast in practice.

Bonomo et al. (2010) showed that the PDTC can be solved in polynomial time for fixed s. In the DTSPMS, s is indeed a fixed input data. However, for the test cases addressed in the literature, where $s \leq 4$ and $|I| \leq 28$, the complexity of the complete enumeration ($O(s^{|I|})$) is much smaller than the complexity of the polynomial algorithm $O(|I|^{s^2})$. Therefore we adopted the complete enumeration approach. This choice is confirmed by our computational experiments, where the execution of our loading procedure never took more than 0.1 CPU second.

It is worth noting that, as in most combined routing and loading problems, the loading aspect makes the routing problem much more difficult. Indeed, the size of DTSPMS instances that can be solved to optimality is dramatically smaller than for the classical TSP. This additional difficulty does not come from the fact that the loading problem is difficult, as this is also true for other well-known combined routing and loading problems, but is rather due to the small size of the set of feasible solutions.

4 Strengthening the Infeasible Path Constraints

In this section, we explain how the linear programming relaxation of model (1)–(6) can be strengthened by the introduction of valid inequalities. The first family of inequalities is based on the classical *tournament constraints* which are often used to improve infeasible path constraints in asymmetric formulations for the TSP (see, e.g., Ascheuer et al., 2000). More specifically, constraint (6) can be strengthened as follows for any load-infeasible pair of paths (Q, F):

$$\sum_{j=1}^{q-1} \sum_{h=j+1}^{q} x^P_{p_j,p_h} + \sum_{j=1}^{f-1} \sum_{h=j+1}^{f} x^D_{d_j,d_h} \leq |A(Q)| + |A(F)| - 1. \tag{7}$$

A graphical representation of constraint (7) is given in Figure 4. The arcs that have been added with respect to Figure 2, hence with respect to constraint (6), derive from the fact that each vertex is visited just once. If one of these additional arcs is used, then two of the original arcs contained in (6) cannot be used and the inequality is still valid.

Looking at Figure 4 suggests a further improvement to the above constraint. It is obvious that the loading violation originates from the clique formed by vertices i, j and k. It is thus independent from the order of visit of the vertices that appear between i and j, or j and k, in one of the two paths. One can thus add reverse arcs among the subset of vertices that belong to the path but not to the clique. This idea is depicted in Figure 5.

More formally, let us denote by C a clique formed by the vertices contained in the undirected version of the precedence graph associated to the two paths. For

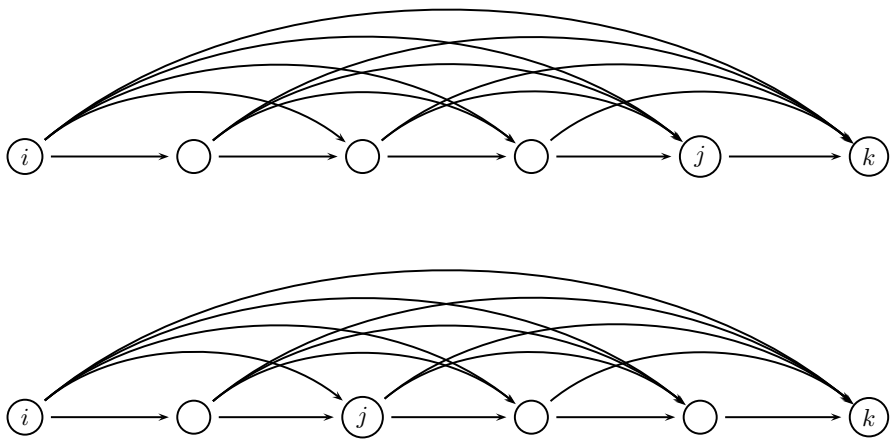

Figure 4: Graphical representation of a tournament constraint.

each vertex $c \in C$, let us denote by S_c^P (resp. S_c^D), the subset of vertices contained in the pickup path (resp. delivery path), and appearing between vertex c and the following vertex belonging to the clique, if any. If $|C| > s$ then the above tournament constraint can be strengthened into the following *lifted tournament constraint* for any load-infeasible pair of paths (Q, F):

$$\sum_{j=1}^{q-1} \sum_{h=j+1}^{q} x^P_{p_j,p_h} + \sum_{c \in C} \sum_{p_j,p_h \in S_c^P : j>h} x^P_{p_j,p_h} + \sum_{j=1}^{f-1} \sum_{h=j+1}^{f} x^D_{d_j,d_h} + \sum_{c \in C} \sum_{d_j,d_h \in S_c^D : j>h} x^D_{d_j,d_h} \leq |A(Q)| + |A(F)| - 1. \quad (8)$$

Note that some subsets S_c^P and S_c^D may be empty, as depicted again in Figure 5. Note also that there may exist paths Q and F which are load-infeasible although there is no clique C having $|C| > s$. In this case, we cannot lift the original tournament constraint.

Another family of valid inequalities derives from the position of a vertex in the stack in which it is loaded. Consider for example Figure 6 and assume for the moment that $sl = n$. The pickup path starts from the depot and ends at vertex k. Suppose $\sigma^P(k)$ is the position of vertex k in this path, i.e., the number of arcs that separate it from the depot along the path. It follows that its distance from the bottom of the stack in which it is loaded is at most $\sigma^P(k)$. Similarly, let us consider a delivery path starting from the delivery depot and ending at vertex k, and let us denote by $\sigma^D(k)$ the position of vertex k in this path. It follows that its distance from the top of the

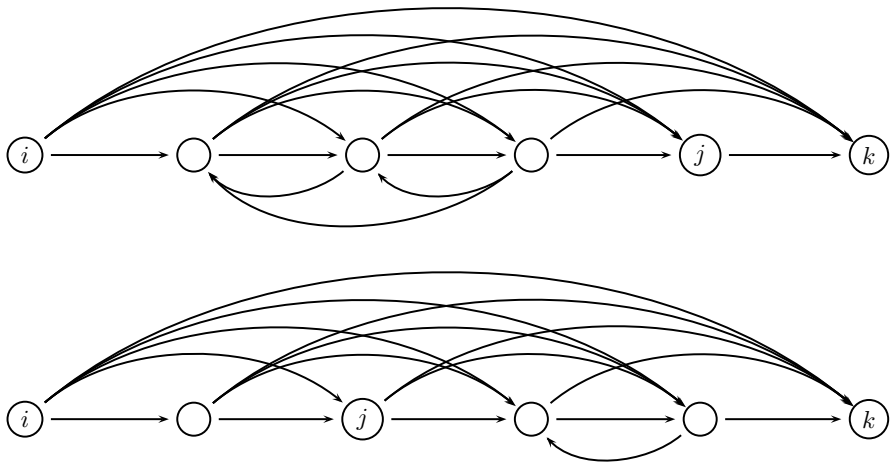

Figure 5: Graphical representation of a lifted tournament constraint.

stack in which it is loaded is at most $\sigma^D(k)$. Hence, if $\sigma^P(k) + \sigma^D(k) \leq l$ then the pair of paths is infeasible. In Figure 6 we have $\sigma^P(k) = 2$ and $\sigma^D(k) = 3$, hence the pair of paths is infeasible for any $l \geq 5$.

In the case where $n < sl$ some positions in the stacks can be left empty and the above condition becomes $\sigma^P(k) + \sigma^D(k) + (sl - n) \leq l$. It follows that, for any load-infeasible pair of paths (Q, F) with both paths starting from 0 and ending at vertex k and such that $\sigma^P(k) + \sigma^D(k) + (sl - n) \leq l$, a valid inequality can be formulated as:

$$\sum_{j=1}^{q-1} \sum_{h=j+1}^{q} x^P_{p_j,p_h} + \sum_{j=3}^{q-1} \sum_{h=2}^{j-1} x^P_{p_j,p_h} + \sum_{j=1}^{f-1} \sum_{h=j+1}^{f} x^D_{d_j,d_h} + \sum_{j=3}^{f-1} \sum_{h=2}^{j-1} x^D_{d_j,d_h} \leq |A(Q)| + |A(F)| - 1. \quad (9)$$

A similar reasoning can also be applied to paths Q and F respectively ending at (instead of starting from) the pickup depot and at the delivery depot. One obtains an inequality very similar to (9), but with indices 0 and k reversed.

Finally, we can apply a reasoning similar to that of the lifted tournament constraints (8) to a pair of paths whose pickup path ends at the pickup depot and whose delivery path starts from the delivery depot. This situation is represented in Figure 7. Let us suppose we have a clique C of size s, hence feasible. We try to prove its infeasibility by enlarging the pickup path from the last vertex up to the pickup depot.

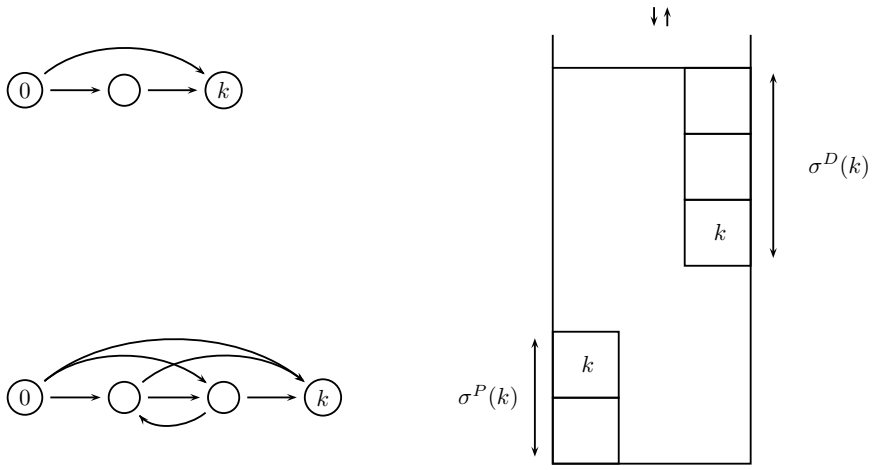

Figure 6: Graphical representation of an infeasible pair of paths starting from the pickup and the delivery depot.

Similarly, we try to enlarge the delivery path backward from the first vertex back to the delivery depot. Suppose that we succeed in doing this and we find two additional sets of vertices, S_c^P in the pickup region and S_c^D in the delivery region. If one of these two sets contains a vertex that is not contained in the other set, then the pair of paths violates the LIFO requirements. Consider for example Figure 7. Vertices i and j are incompatible and have to be loaded into two different stacks. Vertex k is loaded after i and j (it belongs to S_c^P) in the pickup path and unloaded after them in the delivery path (it does not belong to S_c^D). Hence an additional stack is needed to load k, and the example depicted in the Figure 7 is infeasible for $s \leq 2$ stacks.

For any load-infeasible pair of paths (Q, F) with Q ending at 0 and F starting from 0, we thus obtain the following valid inequality:

$$\sum_{j=1}^{q-1} \sum_{h=j+1}^{q} x^P_{p_j,p_h} + \sum_{c \in C} \sum_{p_j,p_h \in S_c^P : j>h} x^P_{p_j,p_h} + \sum_{j=1}^{f-1} \sum_{h=j+1}^{f} x^D_{d_j,d_h} + \sum_{c \in C} \sum_{d_j,d_h \in S_c^D : j>h} x^D_{d_j,d_h} \leq |A(Q)| + |A(F)| - 1, \tag{10}$$

where C is a clique in the incompatibility graph whose cardinality is equal to s, and S_c^T, $T \in \{P, D\}$, is the subset of vertices in the path between vertex c and the following vertex belonging to C.

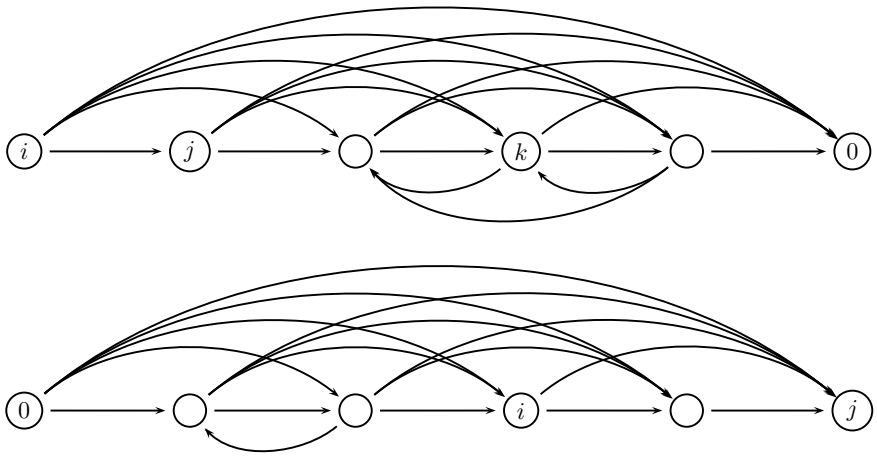

Figure 7: Graphical representation of an infeasible pair of paths ending at the pickup depot and starting from the delivery depot.

5 Branch-and-Cut

We now describe our branch-and-cut algorithm by focusing on the initialization steps, the separation strategies for valid inequalities, and the branching.

5.1 Starting model

At the root node of the enumeration tree we initialize our model with constraints (2) and (3). We set $x_{ii}^T = 0$ for $i = 0, \ldots, n$ and $T \in \{P, D\}$. We also add the subset of inequalities (4) with $|S| = 2$, i.e., $x_{ij}^T + x_{ji}^T \leq 1$ for $i, j = 0, \ldots, n$, $i < j$ and $T \in \{P, D\}$. In the case where $n \geq (s-1)l + 2$, we add the very simple cases of inequality (9) where the paths have unit length, i.e., $x_{0i}^P + x_{0i}^D \leq 1$ and $x_{i0}^P + x_{i0}^D \leq 1$ for $i = 1, \ldots, n$.

5.2 Separation strategy

As is typically done in branch-and-cut algorithms for the TSP, subtour elimination constraints (4) are separated exactly by solving $O(n)$ maximum flow problems.

Infeasible paths constraints (6) are separated exactly as follows. We first identify all possible fractional paths in G^P and G^D starting from any vertex $i = 1, \ldots, n$ and ending at any vertex (initially excluding the depots). We keep only the paths that can possibly lead to a violation, i.e., those for which the sum of the associated

variables is greater than their length minus one. We store these paths in two separate pools, one for the pickup region and one for the delivery region, in non-decreasing value of length, breaking ties by non-increasing sum of the values of the variables associated with the arcs in the path. We then check any possible pair of paths in this order for a possible violation. We check only pairs for which the sum of the associated variables could lead to a violated inequality. The loading violation is checked using the procedure described in Section 3. The separation of the tournament constraints (7) is done in the same way, but the sum of the values of the arc variables is computed by taking the additional arcs into consideration. In our final computational experiments we disregarded the infeasible paths separation and used only the separation of the stronger tournament constraints.

For the lifted tournament constraints (8), we use a heuristic separation procedure. Recall that we invoke the CPM algorithm when we check a possible pair of paths for a loading violation. Any time this procedure returns a clique having a size larger than s, then the path is infeasible and we can add to the model the associated constraint (8), which is stronger than (7). If more than one clique of size larger than s exists, we add a cut for every such clique.

The separation of constraints (9) is performed by identifying all possible (possibly fractional) paths in G^P and G^D starting from both the pickup and the delivery depot, and by checking whether the loading condition $(\sigma^P(k) + \sigma^D(k) + (sl - n) \leq l)$ is violated. The same is done for paths that end at the pickup and the delivery depot.

The separation of constraints (10) is performed heuristically in a similar way to constraints (8). If the CPM algorithm returns a clique with a size exactly equal to s, then we try to find a vertex k that lies on the pickup path between the last vertex of the clique and the depot, and does not lie in the delivery path between the depot and the first vertex of the clique, or vice-versa. If we succeed, a cut is added to the model.

As soon as an infeasible pair of paths is found, we add the corresponding cut to the model and this terminates the separation step. When dealing with symmetric cost matrices (as was the case in our computational experiments), we also add the "reverse" cut, i.e., the cut obtained by reversing the pickup and the delivery paths.

On the basis of computational experiments we decided to use local cuts instead of global ones, i.e., each cut is included in the linear programming relaxation of the current node and of all descendent nodes, but not in that of the other nodes in the tree. We also experimented with other strategies that combined the generation of local and global cuts, but all such strategies proved to be less effective than adding just local cuts.

5.3 Branching

We use the Cplex strong branching strategy to perform branching. In computational experiments, this gave better performance than any other strategy we have tested. In particular, we have tried to extend the pickup path from the pickup depot by always choosing the variable of highest value (one-way extension). We have also attempted extending simultaneously both the pickup and the delivery path by choosing the variable of highest value (two-way extension), and extending the two paths according both to the forward and backward direction (four-way extension). In each case we reduced the graph by removing variables incompatible with the decisions already taken. The strong branching of Cplex turned out to be better than the one-way and two-way extension, and just slightly better than the four-way extension. We have also considered other strategies, such as branching first to 1 and then to 0 or vice-versa, but these did not improve the results.

6 Computational Results

Our branch-and-cut algorithm was implemented in C++ using Cplex 12 in sequential (non-parallel) mode as integer linear programming solver, and all experiments were performed on a 3 GHz Intel Core 2 Duo computer. Several different datasets were considered, all taken from the literature. We refer the reader to the website `www.or.unimore.it/DTSP/dtsp.html` for detailed results on each single instance.

We have first run some experiments on the test instances of Petersen and Madsen (2009) to assess the impact of the valid inequalities described in Section 4. What we have observed is that these inequalities have very little impact on the root node relaxation lower bound, but that they nevertheless improve the performance of the algorithm by reducing the number of nodes explored and the overall computing time for proving optimality. The inequalities were thus used in all further experiments.

In Table 1 we report the results obtained by our algorithm along those of Petersen et al. (2010), Carrabs et al. (2010), and Lusby et al. (2010) on the instances of Petersen and Madsen (2009). Each instance is identified by its name (R05 to R09), the number of stacks s, the height of these stacks l, and the number of requests n. We have used as an initial upper bound the cost of the heuristic solution found by running the heuristic algorithm of Côté et al. (2009), which was provided to us by one of the authors. The initial upper bound supplied to each algorithm, if any, is reported in column UB_0 while columns UB and LB report the final upper and lower bounds, respectively. An asterisk in column *opt* indicates that the instance was solved to optimality. For each algorithm, we report the percentage gap computed as $100(UB - LB)/UB$. We also indicate the CPU time in seconds.

The results show that our algorithm was able to solve 53 of the 60 instances within a one-hour time limit. For the remaining instances, the final optimality gap is at most 2.95%. In comparison, the algorithm of Petersen et al. (2010) was only able to solve 30 instances and the optimality gap for the unsolved instances sometimes exceeds 10%. Carrabs et al. (2010) only reported results on instances with two stacks while Lusby et al. (2010) reported results on instances with a number of requests between 10 and 15. In terms of computing speed, our algorithm is also considerably better. For the instances that were solved to optimality by at least one other method, our branch-and-cut algorithm is often faster by one order of magnitude. Finally, we should also point out that the CPU time limit imposed by Lusby et al. (2010) was three hours.

In Table 2 we report similar results on a set of 81 instances which were introduced in the context of the PDTSPL (see Cordeau et al., 2010) and were later adapted for the DTSPMS by Petersen et al. (2010). Our algorithm was able to solve 69 instances of these instances, including all those with up to 19 requests and some with 21, 23 and 25 requests. The optimality gap for the unsolved instances is at most 3.06%. In comparison, the algorithm of Petersen et al. (2010) could solve 46 instances and failed on one with 15 requests.

Table 3 reports more detailed results on the Petersen and Madsen (2009) instances with just two stacks and 10, 12 or 14 requests. Using a maximum computing time of 3 hours, as did Lusby et al. (2010) and Carrabs et al. (2010), we were able to solve all 60 instances, whereas the other algorithms could solve 44 and 56 instances, respectively. For most of the instances that were solved by at least one of the other algorithms, our branch-and-cut is again much faster.

Finally, Table 4 reports summary results for the full set of 220 instances introduced by Petersen and Madsen (2009). In this table, the instances are divided into 11 groups of 20 instances each. We indicate the number of instances that could be solved to optimality by each algorithm as well as the average optimality gaps and the average computing time for each group. A maximum of three hours of computing time was allowed for solving each instance. Again, we can see that our new method outperforms the two existing algorithms by solving more instances and by requiring far less computing time. Our algorithm was capable of solving to optimality 192 of the instances and the average computing time is about half an hour. The maximum optimality gap for the unsolved instances is 3.15%. The average optimality gap over all instances (including those solved to optimality) is just 0.16%. We finally note that an increase in the height of the stacks has a much larger impact on the difficulty of the problem than an increase in the number of stacks.

7 Conclusions

This paper has introduced a new branch-and-cut algorithm for the DTSPMS. This algorithm relies on a compact formulation of the problem and the addition of valid inequalities that can be separated efficiently. Computational experiments performed on a large set of test instances show that this new algorithm outperforms existing exact methods in terms of the problem size that can be solved and of the required computing time.

Acknowledgements

This work was partly supported by the Italian Ministero dell'Istruzione, dell'Università e della Ricerca (MIUR) and by the Canadian Natural Sciences and Engineering Research Council under grant 227837-09. This support is gratefully acknowledged. We are also thankful to Jean-François Côté for fruitful discussions and for providing us with the executable of the algorithm introduced by Côté et al. (2009).

References

N. Ascheuer, M. Fischetti, and M. Grötschel. A polyhedral study of the asymmetric traveling salesman problem with time windows. *Networks*, 36:69–79, 2000.

F. Bonomo, S. Mattia, and G. Oriolo. Bounded coloring of co-comparability graphs and the pickup and delivery tour combination problem. Technical Report 6, DEIS, Sapienza Università di Roma, Italy, 2010.

F. Carrabs, R. Cerulli, and J.-F. Cordeau. An additive branch-and-bound algorithm for the pickup and delivery traveling salesman problem with LIFO or FIFO loading. *INFOR*, 45:223–238, 2007a.

F. Carrabs, J.-F. Cordeau, and G. Laporte. Variable neighbourhood search for the pickup and delivery traveling salesman problem with LIFO loading. *INFORMS Journal on Computing*, 19:618–632, 2007b.

F. Carrabs, R. Cerulli, and M.G. Speranza. A branch-and-bound algorithm for the double TSP with two stacks. Technical Report 4, DMI, Università di Salerno, Italy, http://www.dmi.unisa.it/people/carrabs/www/, 2010.

M. Casazza, A. Ceselli, and M. Nunkesser. Efficient algorithms for the double TSP with multiple stacks. In *Proceedings of 8th Cologne-Twente Workshop on Graphs and Combinatorial Optimization (CTW09)*, pages 7–10, Paris, 2009.

J.-F. Cordeau, G. Laporte, J.-Y. Potvin, and M.W.P. Savelsbergh. Transportation on demand. In C. Barnhart and G. Laporte, editors, *Transportation*, Handbooks in Operations Research and Management Science 14, pages 429–466. Elsevier, Amsterdam, 2007.

J.-F. Cordeau, M. Iori, G. Laporte, and J.J. Salazar-Gonzalez. Branch-and-cut for the pickup and delivery traveling salesman problem with LIFO loading. *Networks*, 55:46–59, 2010.

J.-F. Côté, M. Gendreau, and J.-Y. Potvin. Large neighborhood search for the single vehicle pickup and delivery problem with multiple loading stacks. Technical Report CIRRELT-2009-47, University of Montreal, 2009.

A. Felipe, M.T. Ortuno, and G. Tirado. The double traveling salesman problem with multiple stacks: A variable neighborhood search approach. *Computers & Operations Research*, 36:2983–2993, 2009a.

A. Felipe, M.T. Ortuno, and G. Tirado. New neighborhood structures for the double traveling salesman problem with multiple stacks. *TOP*, 17:190–213, 2009b.

P. Hansen, A. Hertz, and J. Kuplinsky. Bounded vertex colorings of graphs. *Discrete Mathematics*, 111:305–312, 1993.

M. Iori and S. Martello. Routing problems with loading constraints. *TOP*, 18:4–27, 2010.

J. Kelley. Critical path planning and scheduling: Mathematical basis. *Operations Research*, 9:296–320, 1961.

R.M. Lusby, J. Larsen, M. Ehrgott, and D. Ryan. An exact method for the double TSP with multiple stacks. *International Transactions on Operations Research*, 17: 637–652, 2010.

H.L. Petersen and O.B.G. Madsen. The double travelling salesman problem with multiple stacks. *European Journal of Operational Research*, 198:139–147, 2009.

H.L. Petersen, C. Archetti, and M.G. Speranza. Exact solutions to the double travelling salesman problem with multiple stacks. *Networks*, 2010. To appear.

S. Toulouse. Approximability of the multiple stack TSP. *Electronic Notes in Discrete Mathematics*, 36:813–820, 2010.

S. Toulouse and R. Wolfler Calvo. On the complexity of the multiple stack TSP, kSTSP. *Lecture Notes in Computer Science, Theory and Applications of Models of Computation*, 5532/2009:360–369, 2009.

Table 1: Comparison on 60 benchmark instances (1 CPU hour)

				Petersen et al. (2010) Pentium 4, 2.8 GHz				Carrabs et al. (2010) Intel Core 2, 2.33 GHz			Lusby et al. (2010) Dell, 1.6 GHz				Branch-and-Cut Intel Core 2, 3 GHz					
inst.	s	l	n	UB_0	%gap	opt	sec	UB	opt	sec	UB	%gap	opt	sec	UB_0	UB	LB	%gap	opt	sec
R05	2	4	8	501	0.00%	*	0	501	*	0.25					501	501	501	0.00%	*	0.0
R06	2	4	8	694	0.00%	*	31	694	*	0.51					694	694	694	0.00%	*	0.1
R07	2	4	8	487	0.00%	*	27	487	*	0.56					487	487	487	0.00%	*	0.1
R08	2	4	8	642	0.00%	*	38	642	*	0.57					642	642	642	0.00%	*	0.1
R09	2	4	8	558	0.00%	*	17	558	*	0.76					558	558	558	0.00%	*	0.0
R05	2	5	10	546	0.00%	*	196	546	*	2.28	546	0.00%	*	4	546	546	546	0.00%	*	0.3
R06	2	5	10	774	0.00%	*	678	774	*	5.66	774	0.00%	*	5	774	774	774	0.00%	*	1.6
R07	2	5	10	547	0.00%	*	115	547	*	1.60	547	0.00%	*	1	547	547	547	0.00%	*	0.2
R08	2	5	10	670	0.00%	*	392	670	*	2.00	670	0.00%	*	5	670	670	670	0.00%	*	0.7
R09	2	5	10	610	0.00%	*	44	610	*	4.65	610	0.00%	*	1	610	610	610	0.00%	*	0.1
R05	2	6	12	631	7.92%		3600	631	*	172.26	631	0.00%	*	2126	631	631	631	0.00%	*	176.8
R06	2	6	12	793	2.77%		3600	793	*	111.94	793	0.00%	*	310	793	793	793	0.00%	*	3.3
R07	2	6	12	593	4.05%		3600	593	*	61.15	593	0.00%	*	485	593	593	593	0.00%	*	8.5
R08	2	6	12	749	4.65%		3600	749	*	74.96	749	0.13%		10823	749	749	749	0.00%	*	30.1
R09	2	6	12	692	0.00%	*	1680	692	*	219.22	692	0.00%	*	45	692	692	692	0.00%	*	1.8
R05	2	7	14	775	7.08%		3600	775	*	3052.79	775	1.98%		10807	775	775	775	0.00%	*	392.6
R06	2	7	14	824	5.22%		3600	824	*	2655.48	824	0.73%		10814	824	824	824	0.00%	*	33.4
R07	2	7	14	697	6.95%		3600	697	*	341.83	697	2.20%		10822	697	697	697	0.00%	*	48.4
R08	2	7	14	824	8.01%		3600	825		3604.46	831	5.73%		10815	824	824	824	0.00%	*	672.5
R09	2	7	14	739	1.53%		3600	739	*	1225.67	739	0.00%	*	211	739	739	739	0.00%	*	5.3
R05	3	4	12	567	0.00%	*	7				567	0.00%	*	1	567	567	567	0.00%	*	0.1
R06	3	4	12	747	0.00%	*	15				747	0.00%	*	2	747	747	747	0.00%	*	0.1
R07	3	4	12	557	0.00%	*	90				557	0.00%	*	5	557	557	557	0.00%	*	0.3
R08	3	4	12	690	0.00%	*	7				690	0.00%	*	2	690	690	690	0.00%	*	0.0
R09	3	4	12	672	0.00%	*	6				669	0.00%	*	1	669	669	669	0.00%	*	0.0
R05	3	5	15	737	0.00%	*	2442				737	0.00%	*	56	737	737	737	0.00%	*	2.6
R06	3	5	15	836	0.00%	*	1815				836	0.00%	*	47	836	836	836	0.00%	*	2.2
R07	3	5	15	690	2.39%		3600				690	0.00%	*	415	690	690	690	0.00%	*	3.6
R08	3	5	15	826	0.00%	*	2028				826	0.00%	*	73	826	826	826	0.00%	*	3.1
R09	3	5	15	768	0.00%	*	666				768	0.00%	*	29	768	768	768	0.00%	*	2.3
R05	3	6	18	833	6.89%		3600								804	804	804	0.00%	*	23.8
R06	3	6	18	871	3.56%		3600								866	866	866	0.00%	*	144.6
R07	3	6	18	758	8.08%		3600								752	752	752	0.00%	*	119.9
R08	3	6	18	864	4.61%		3600								864	864	864	0.00%	*	134.3
R09	3	6	18	796	0.00%	*	1995								774	774	774	0.00%	*	1.5
R05	3	7	21	900	7.64%		3600								875	875	862	1.49%		3600.4
R06	3	7	21	949	10.41%		3600								901	901	901	0.00%	*	1755.8
R07	3	7	21	841	9.45%		3600								836	836	826	1.20%		3600.4
R08	3	7	21	916	9.53%		3600								888	888	881	0.79%		3600.6
R09	3	7	21	891	10.21%		3600								827	827	827	0.00%	*	72.1
R05	4	4	16	750	0.00%	*	1271								744	744	744	0.00%	*	2.3
R06	4	4	16	841	0.00%	*	281								821	821	821	0.00%	*	0.5
R07	4	4	16	673	0.00%	*	3								673	673	673	0.00%	*	0.0
R08	4	4	16	819	0.00%	*	123								815	815	815	0.00%	*	0.4
R09	4	4	16	774	0.00%	*	1								755	755	755	0.00%	*	0.0
R05	4	5	20	856	5.57%		3600								825	825	825	0.00%	*	26.4
R06	4	5	20	894	0.00%	*	2347								859	859	859	0.00%	*	0.8
R07	4	5	20	795	0.00%	*	3542								763	763	763	0.00%	*	0.5
R08	4	5	20	853	0.00%	*	1614								841	841	841	0.00%	*	4.2
R09	4	5	20	818	0.00%	*	4								796	796	796	0.00%	*	0.1
R05	4	6	24	933	9.62%		3600								867	867	867	0.00%	*	474.5
R06	4	6	24	975	10.77%		3600								898	898	898	0.00%	*	128.5
R07	4	6	24	916	11.90%		3600								831	831	831	0.00%	*	14.4
R08	4	6	24	924	5.25%		3600								904	904	904	0.00%	*	98.4
R09	4	6	24	882	6.04%		3600								863	863	861	0.23%		3600.5
R05	4	7	28	984	11.88%		3600								895	895	895	0.00%	*	2023.7
R06	4	7	28	1034	10.93%		3600								964	964	951	1.35%		3600.4
R07	4	7	28	1002	12.44%		3600								948	948	920	2.95%		3600.4
R08	4	7	28	1088	15.26%		3600								949	949	949	0.00%	*	258.2
R09	4	7	28	975	9.63%		3600								918	918	906	1.31%		3600.4
Totals/Averages					3.84%	*30*	2157.9											0.16%	*53*	531.3

Table 2: Comparison on 81 instances from the PDTSPL literature (1 CPU hour)

				Petersen et al. (2010) Pentium 4, 2.8 GHz					Branch-and-Cut Intel Core 2, 3 GHz				
inst.	s	l	n	UB	LB	%gap	opt	sec	UB	LB	%gap	opt	sec
a280	3	3	9	585	585	0.00%	*	2	585	585	0.00%	*	0.1
	3	4	11	654	654	0.00%	*	51	654	654	0.00%	*	0.2
	3	5	13	696	696	0.00%	*	19	696	696	0.00%	*	0.1
	3	5	15	792	792	0.00%	*	31	792	792	0.00%	*	0.1
	3	6	17	945	945	0.00%	*	2277	945	945	0.00%	*	0.4
	3	7	19	-	1017	-		3600	1024	1024	0.00%	*	0.7
	3	7	21	1127	1091	3.21%		3600	1103	1103	0.00%	*	2.4
	3	8	23	-	1160.5	-		3600	1179	1179	0.00%	*	11.7
	3	9	25	-	1192	-		3600	1219	1219	0.00%	*	49.2
att532	3	3	9	5361	5361	0.00%	*	2	5361	5361	0.00%	*	0.0
	3	4	11	6399	6399	0.00%	*	23	6399	6399	0.00%	*	0.0
	3	5	13	7261	7261	0.00%	*	102	7261	7261	0.00%	*	0.1
	3	5	15	7562	7562	0.00%	*	320	7562	7562	0.00%	*	0.3
	3	6	17	11369	7737	31.95%		3600	7863	7863	0.00%	*	1.5
	3	7	19	11413	7972	30.16%		3600	8208	8208	0.00%	*	14.1
	3	7	21	13218	12230	7.48%		3600	12639	12639	0.00%	*	86.1
	3	8	23	-	12530	-		3600	13006	13006	0.00%	*	303.7
	3	9	25	-	15709	-		3600	16214	16214	0.00%	*	1181.7
brd14051	3	3	9	7897	7897	0.00%	*	0	7897	7897	0.00%	*	0.0
	3	4	11	8064	8064	0.00%	*	1	8064	8064	0.00%	*	0.0
	3	5	13	8079	8079	0.00%	*	41	8079	8079	0.00%	*	0.1
	3	5	15	8196	8196	0.00%	*	3	8196	8196	0.00%	*	0.2
	3	6	17	8300	8226	0.89%		3600	8252	8252	0.00%	*	70.3
	3	7	19	8434	8394	0.48%		3600	8419	8419	0.00%	*	62.8
	3	7	21	9109	8400	7.79%		3600	8442	8442	0.00%	*	130.0
	3	8	23	-	8499	-		3600	8560	8551	0.11%		3600.4
	3	9	25	-	8513	-		3600	8644	8588	0.65%		3600.6
d15112	3	3	9	93597	93597	0.00%	*	28	93597	93597	0.00%	*	0.1
	3	4	11	100489	100489	0.00%	*	39	100489	100489	0.00%	*	0.2
	3	5	13	108574	108574	0.00%	*	211	108574	108574	0.00%	*	0.4
	3	5	15	130297	124692	4.30%		3600	127814	127814	0.00%	*	3.5
	3	6	17	141408	126627	10.45%		3600	131421	131421	0.00%	*	47.7
	3	7	19	-	130153	-		3600	136488	136488	0.00%	*	504.0
	3	7	21	188222	132034	29.85%		3600	139965	138437	1.09%		3600.3
	3	8	23	-	133448	-		3600	141404	139508	1.34%		3600.4
	3	9	25	-	138886	-		3600	149772	145188	3.06%		3600.4
d18512	3	3	9	7951	7951	0.00%	*	1	7951	7951	0.00%	*	0.1
	3	4	11	8023	8023	0.00%	*	1	8023	8023	0.00%	*	0.0
	3	5	13	8034	8034	0.00%	*	6	8034	8034	0.00%	*	0.0
	3	5	15	8098	8098	0.00%	*	19	8098	8098	0.00%	*	0.0
	3	6	17	8567	8124	5.17%		3600	8151	8151	0.00%	*	54.7
	3	7	19	-	8292	-		3600	8327	8327	0.00%	*	120.2
	3	7	21	10664	8425	21.00%		3600	8482	8482	0.00%	*	1072.3
	3	8	23	-	8476	-		3600	8555	8529	0.30%		3600.4
	3	9	25	-	8556	-		3600	8672	8607	0.75%		3600.4
fnl4461	3	3	9	3387	3387	0.00%	*	1	3387	3387	0.00%	*	0.1
	3	4	11	3430	3430	0.00%	*	9	3430	3430	0.00%	*	0.0
	3	5	13	3628	3628	0.00%	*	185	3628	3628	0.00%	*	1.0
	3	5	15	3796	3796	0.00%	*	192	3796	3796	0.00%	*	0.3
	3	6	17	3853	3837	0.42%		3600	3853	3853	0.00%	*	6.6
	3	7	19	5344	3981	25.52%		3600	4027	4027	0.00%	*	92.5
	3	7	21	4589	4058	11.58%		3600	4147	4147	0.00%	*	813.1
	3	8	23	-	4170	-		3600	4315	4273	0.97%		3600.4
	3	9	25	-	4253	-		3600	4427	4349	1.76%		3600.6
nrw1379	3	3	9	4572	4572	0.00%	*	3	4572	4572	0.00%	*	0.2
	3	4	11	4733	4733	0.00%	*	17	4733	4733	0.00%	*	0.0
	3	5	13	4872	4872	0.00%	*	273	4872	4872	0.00%	*	0.9
	3	5	15	4984	4984	0.00%	*	1230	4984	4984	0.00%	*	2.2
	3	6	17	5355	5195	2.99%		3600	5212	5212	0.00%	*	1.8
	3	7	19	-	5245	-		3600	5320	5320	0.00%	*	1099.6
	3	7	21	6114	5434	11.12%		3600	5543	5535	0.14%		3600.4
	3	8	23	-	5481	-		3600	5592	5582	0.18%		3600.4
	3	9	25	-	5862	-		-	6056	5961	1.57%		3600.5
pr1002	3	3	9	21498	21498	0.00%	*	0	21498	21498	0.00%	*	0.0
	3	4	11	22977	22977	0.00%	*	15	22977	22977	0.00%	*	0.1
	3	5	13	25087	25087	0.00%	*	184	25087	25087	0.00%	*	0.1
	3	5	15	25899	25899	0.00%	*	929	25899	25899	0.00%	*	0.3
	3	6	17	27246	27246	0.00%	*	731	27246	27246	0.00%	*	0.2
	3	7	19	28196	28196	0.00%	*	1733	28196	28196	0.00%	*	0.5
	3	7	21	29875	29875	0.00%	*	5	29875	29875	0.00%	*	0.1
	3	8	23	31463	31463	0.00%	*	133	31463	31463	0.00%	*	0.1
	3	9	25	32319	32319	0.00%	*	5	32319	32319	0.00%	*	0.1
ts225	3	3	9	34000	34000	0.00%	*	0	34000	34000	0.00%	*	0.0
	3	4	11	43000	43000	0.00%	*	443	43000	43000	0.00%	*	0.2
	3	5	13	48440	48400	0.00%	*	2	48440	48440	0.00%	*	0.0
	3	5	15	50580	50580	0.00%	*	4	50580	50580	0.00%	*	0.1
	3	6	17	50881	50881	0.00%	*	2	50881	50881	0.00%	*	0.1
	3	7	19	51371	51371	0.00%	*	17	51371	51371	0.00%	*	0.1
	3	7	21	52322	52322	0.00%	*	8	52322	52322	0.00%	*	0.1
	3	8	23	54460	54460	0.00%	*	6	54460	54460	0.00%	*	0.1
	3	9	25	62688	62688	0.00%	*	808	62688	62688	0.00%	*	1.1
Totals/Averages							*46*				0.15%	*69*	604.3

Table 3: Further comparison on 60 instances with two stacks (3 CPU hours)

				Lusby et al. (2010) Dell, 1.6 GHz			Carrabs et al. (2010) Intel Core 2, 2.33 GHz		Branch-and-Cut Intel Core 2, 3 GHz				
inst.	s	l	n	%gap	opt	sec	opt	sec	UB	LB	%gap	opt	sec
R00	2	5	10	0.00%	*	5	*	5.12	680	680	0.00%	*	0.6
R01	2	5	10	0.00%	*	3	*	4.11	704	704	0.00%	*	0.5
R02	2	5	10	0.00%	*	6	*	7.57	629	629	0.00%	*	1.6
R03	2	5	10	0.00%	*	1	*	0.98	610	610	0.00%	*	0.1
R04	2	5	10	0.00%	*	3	*	1.28	614	614	0.00%	*	0.2
R05	2	5	10	0.00%	*	4	*	2.29	546	546	0.00%	*	0.3
R06	2	5	10	0.00%	*	5	*	5.72	774	774	0.00%	*	1.6
R07	2	5	10	0.00%	*	1	*	1.61	547	547	0.00%	*	0.2
R08	2	5	10	0.00%	*	5	*	2.02	670	670	0.00%	*	0.7
R09	2	5	10	0.00%	*	1	*	4.68	610	610	0.00%	*	0.1
R10	2	5	10	0.00%	*	7	*	0.88	624	624	0.00%	*	0.4
R11	2	5	10	0.00%	*	2	*	0.56	536	536	0.00%	*	0.1
R12	2	5	10	0.00%	*	3	*	1.72	678	678	0.00%	*	0.3
R13	2	5	10	0.00%	*	2	*	1.30	654	654	0.00%	*	0.2
R14	2	5	10	0.00%	*	13	*	4.78	603	603	0.00%	*	1.3
R15	2	5	10	0.00%	*	2	*	2.52	586	586	0.00%	*	0.3
R16	2	5	10	0.00%	*	114	*	0.99	535	535	0.00%	*	3.8
R17	2	5	10	0.00%	*	11	*	4.52	729	729	0.00%	*	1.1
R18	2	5	10	0.00%	*	1	*	0.58	616	616	0.00%	*	0.1
R19	2	5	10	0.00%	*	1	*	0.58	650	650	0.00%	*	0.1
R00	2	6	12	0.00%	*	142	*	108.48	726	726	0.00%	*	8.3
R01	2	6	12	0.00%	*	17	*	68.51	741	741	0.00%	*	1.8
R02	2	6	12	0.00%	*	2432	*	197.68	660	660	0.00%	*	42.8
R03	2	6	12	0.00%	*	4	*	3.43	690	690	0.00%	*	0.2
R04	2	6	12	0.00%	*	1151	*	62.28	659	659	0.00%	*	40.7
R05	2	6	12	0.00%	*	2126	*	173.91	631	631	0.00%	*	176.8
R06	2	6	12	0.00%	*	310	*	109.72	793	793	0.00%	*	3.3
R07	2	6	12	0.00%	*	485	*	61.80	593	593	0.00%	*	8.5
R08	2	6	12	0.13%		10823	*	75.36	749	749	0.00%	*	30.1
R09	2	6	12	0.00%	*	45	*	220.29	692	692	0.00%	*	1.8
R10	2	6	12	0.00%	*	2452	*	16.62	663	663	0.00%	*	12.0
R11	2	6	12	0.00%	*	356	*	53.16	625	625	0.00%	*	17.9
R12	2	6	12	0.00%	*	7	*	7.32	741	741	0.00%	*	0.3
R13	2	6	12	0.00%	*	16	*	8.10	694	694	0.00%	*	1.4
R14	2	6	12	0.00%	*	205	*	110.49	680	680	0.00%	*	4.2
R15	2	6	12	0.00%	*	306	*	85.37	628	628	0.00%	*	20.6
R16	2	6	12	0.00%	*	3537	*	4.04	610	610	0.00%	*	25.4
R17	2	6	12	0.00%	*	3832	*	551.97	780	780	0.00%	*	92.3
R18	2	6	12	0.00%	*	16	*	14.00	735	735	0.00%	*	0.7
R19	2	6	12	0.00%	*	171	*	150.80	789	789	0.00%	*	9.0
R00	2	7	14	1.18%		10838	*	7592.21	774	774	0.00%	*	156.6
R01	2	7	14	0.00%	*	653	*	1787.94	761	761	0.00%	*	41.9
R02	2	7	14	1.47%		10863	*	5638.48	690	690	0.00%	*	125.5
R03	2	7	14	0.13%		10807	*	423.42	791	791	0.00%	*	24.1
R04	2	7	14	6.47%		10805	*	6437.22	756	756	0.00%	*	3815.1
R05	2	7	14	1.98%		10807	*	3040.04	775	775	0.00%	*	392.6
R06	2	7	14	0.73%		10814	*	2657.65	824	824	0.00%	*	33.4
R07	2	7	14	2.20%		10822	*	339.85	697	697	0.00%	*	48.4
R08	2	7	14	5.73%		10815		10802.00	824	824	0.00%	*	672.5
R09	2	7	14	0.00%	*	211	*	1209.25	739	739	0.00%	*	5.3
R10	2	7	14	2.09%		10845	*	3924.84	733	733	0.00%	*	390.8
R11	2	7	14	2.84%		10815	*	3658.66	725	725	0.00%	*	989.0
R12	2	7	14	1.01%		10814	*	343.56	803	803	0.00%	*	86.5
R13	2	7	14	0.00%	*	1499	*	240.62	746	746	0.00%	*	26.4
R14	2	7	14	-		10841		10802.00	765	765	0.00%	*	10134.9
R15	2	7	14	0.00%	*	1152	*	436.30	765	765	0.00%	*	22.1
R16	2	7	14	0.88%		10826	*	284.46	685	685	0.00%	*	66.6
R17	2	7	14	4.41%		10860		10802.00	818	818	0.00%	*	956.7
R18	2	7	14	0.00%	*	3607	*	757.32	774	774	0.00%	*	31.2
R19	2	7	14	1.81%		10839		10802.00	836	836	0.00%	*	149.3
Totals/Averages				0.56%	*44*	3302.60	*56*	1401.95			0.00%	*60*	311.3

Table 4: Aggregate comparison on 220 benchmark instances (3 CPU hours)

			Lusby et al. (2010) Dell, 1.6 GHz			Carrabs et al. (2010) Intel Core 2, 2.33 GHz		Branch-and-Cut Intel Core 2, 3 GHz		
s	l	n	%gap	opt	sec	opt	sec	%gap	opt	sec
2	5	10	0.00%	20	9.5	20	2.7	0.00%	20	0.7
2	6	12	0.01%	19	1421.7	20	104.2	0.00%	20	24.9
2	7	14	1.73%	5	8476.7	16	4099.0	0.00%	20	908.4
3	4	12	0.00%	20	4.0			0.00%	20	0.2
3	5	15	0.00%	20	492.2			0.00%	20	13.5
3	6	18						0.00%	20	275.8
3	7	21						0.78%	8	8001.1
4	4	16						0.00%	20	0.7
4	5	20						0.00%	20	15.3
4	6	24						0.00%	20	1287.6
4	7	28						0.97%	4	9264.9
Totals/Averages				*84*			*56*	0.16%	*192*	1799.4