A Branch-and-Bound Algorithm for the Double TSP with Two Stacks

Francesco Carrabs*, Raffaele Cerulli*, M. Grazia Speranza†

Abstract

This paper studies the double traveling salesman problem with two stacks. A number of requests have to be served where each request consists in the pickup and delivery of an item. All the pickup operations have to be performed before any delivery can take place. A single vehicle is available that starts from a depot, performs all the pickup operations and returns to the depot. Then, it performs all the delivery operations and returns to the depot. The items are loaded in two stacks, each served independently from the other with a LIFO policy. The objective is the minimization of the total cost of the pickup and delivery tours. We propose a branch-and-bound approach to solve the problem. The algorithm uses properties of the problem both to tighten the lower bounds and to avoid the exploration of redundant subtrees. Computational results performed on benchmark instances reveal that the algorithm outperforms the other exact approaches for this problem.

Keywords: Double traveling salesman problem; Pickup and delivery problems; Branch-and-Bound; LIFO; Multiple stacks.

1. Introduction

Most of the literature on vehicle routing problems studies the problem of creating routes that minimize the routing costs. In practice, the problem of how to load the vehicles is considered to have an important impact on the costs. In a sequence of delivery operations, the driver may need to spend time in rearranging the content of the vehicle if the items to be delivered have not been properly loaded. The loading and the routing problems influence each other. If the loading problem is solved independently of the routing problem, it may happen that the items are loaded in such a way that the driver has to rearrange them before any delivery operation can take place. Thus, although the route may be shortest in terms of traveling cost, the solution obtained may not be the best possible. In the case of a pickup and delivery problem, the loading problem becomes even more relevant.

Recently, although the literature is still quite limited, a number of contributions appeared that treat combined routing and loading problems. Several different combinations of the routing and loading problems have been considered. Mainly heuristic solution approaches have been proposed but some contributions are also available on exact approaches. Heuristics for the vehicle routing problem with two dimensional loading constraints have been proposed in [15] and [18] . An exact approach is available for the same problem [19]. A heuristic for the extension of the problem to the case of three dimensional loading constraints has been presented in [16]. Other loading constraints for the vehicle routing problem have been studied in [9], [17]

*Department of Mathematics and Computer Science, University of Salerno, Italy. fcarrabs@unisa.it, raffaele@unisa.it
†Department of Quantitative Methods, University of Brescia, Italy. speranza@eco.unibs.it

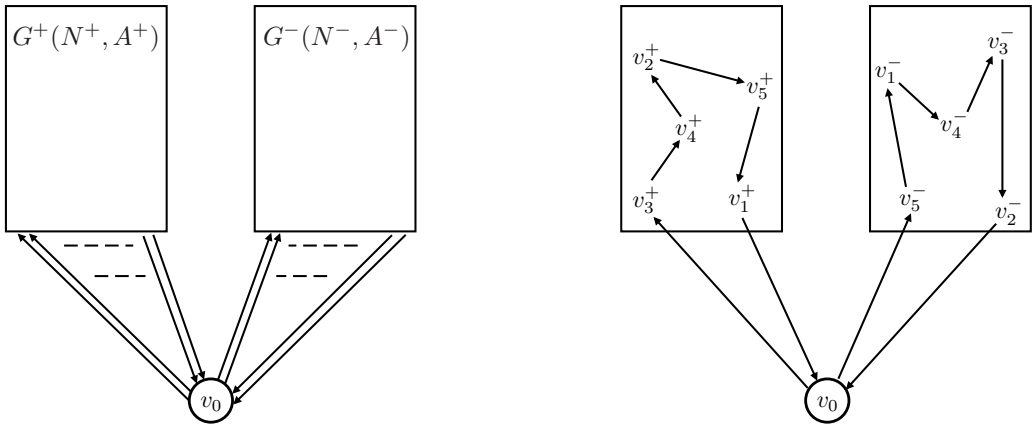
and [25]. Whereas in [9] and [17] a heuristic has been proposed, in [25] both a heuristic and an exact solution approach have been presented.

Pick-up and delivery routing problems combined with loading constraints have also received attention. However, in this case the case of one vehicle only has been considered. Heuristic and exact solution approaches have been proposed for the pickup and delivery traveling salesman problem (TSP) with different types of loading constrains in [2], [4], [6], [7] and [11].

In this paper we consider the Double Traveling Salesman Problem with Two Stacks (DTSP2S). This is a pickup and delivery problem where a set of requests has to be satisfied by a vehicle at minimum cost. Satisfying a request means that an item has to be picked up at a customer and delivered to a destination customer. All the pickup operations have to be performed before any delivery operation can take place. A vehicle is available to perform the pickup and the delivery operations, respectively. The vehicle starts from the depot, visits all the pickup customers and returns to the depot. Then, it visits all the delivery customers and returns to the depot. The vehicle carries a container that is structured in two horizontal stacks. Each stack can accommodate a given maximum number of items and is served with a Last-In-First-Out (LIFO) policy. When an item is picked up, it can be loaded into any of the two stacks that has still space available. In the delivery phase, each of the two stacks must be emptied according to a LIFO policy. No repacking is allowed between the pickup and the delivery phases. Thus, whenever a delivery operation takes place, at most two items are available to be unloaded, one per stack. If a stack is empty only one item is available. The problem consists in identifying a pickup tour, a loading policy and a delivery tour in such a way that the sum of the costs of the two tours is minimum.

The DTSP2S is derived from the more general double traveling salesman problem with multiple stacks (DTSPMS). The DTSPMS is a generalization of the TSP with pickup and delivery (TSPPD) and is similar to the TSP with pickup and delivery and LIFO constraints (TSPPDL). Indeed, the latter problem can be seen as a DTSPMS where the vehicle contains a single stack with infinite capacity and no precedence constraint is set between the pickup and the delivery operations. For the TSPPDL, a variable neighborhood search (VNS) heuristic has been introduced in [4], while two exact approaches have been proposed in [3] and [7]. The results obtained so far suggest that the introduction of multiple stacks makes the problem much harder to solve.

Although the DTSPMS has been only recently introduced, in the last years it has received a growing interest, probably because, in spite of the simple structure, it is computationally very challenging. The problem was introduced in [22], where the authors proposed a mathematical model and developed two neighborhoods used in four metaheuristics (Iterated Local Search, Tabu Search, Simulated Annealing and Large Neighborhood Search). In [12] the authors introduced four new neighborhoods that, together with the ones proposed in [22], were embedded in a new VNS metaheuristic. Computational results show that the new VNS metaheuristic outperforms previous metaheuristics presented in the literature. Finally, a large neighborhood search heuristic has been proposed in [8]. Many benchmark instances for both the TSPPDL and DTSPMS are used by the authors to prove that their algorithm overcomes the ones proposed in the literature. An interesting analysis of basic subproblems of the DTSPMS has been carried out in [5] and [24]. To the best of our knowledge only two exact solution approaches for the DTSPMS have been presented([20] and [21]). In [20] an exact solution approach based on matching k-best TSP solutions for each of the separate pickup and delivery tours was proposed. In [21] the authors introduced various formulations of the problem and solved them with a branch-and-cut approach. It is very interesting to notice that for both these previous exact approaches, the difficulty of an instance depends on the capacity of the stacks, given the number of requests. Since the capacity is equal to $\lceil requests/stacks \rceil$, the performance of the algorithms improves, given the number of requests, when the number of stacks increases. This is probably due to the fact that the construction of the tours becomes less constrained. On the contrary, the performance of the algorithm we propose in this paper improves when the number of stacks decreases, because, thanks to the precedences



(a) (b)

Figure 1: The graphs G^+ and G^- (a) and a solution (b).

generated inside the same stack, the branch-and-bound we present is able to find tighter lower bounds. The number of stacks considered in the benchmark instances for the DTSPMS is two or three, which are the most realistic cases. In this paper we focus on the case of two stacks and propose an additive branch-and-bound solution approach that improves previously known results on benchmark instances. While the theorems, properties and strategies introduced in this paper hold for any number of stacks, preliminary tests showed that the proposed branch-and-bound approach is not competitive with the other known approaches in the case of three stacks.

The remainder of the paper is organized as follows. In Section 2 we formally define the problem with two stacks. The additive branch-and-bound algorithm is described in Section 3. Finally, the computational results are presented in Section 4 and some concluding remarks are given in Section 5.

2. Definitions and Notation

In the DTSP2S we have n requests, each composed by a pickup vertex v^+ and a delivery vertex v^-. In order to satisfy a request we have to load an item in v^+ and unload it in v^-. We define $N^+ = \{v_1^+, \ldots, v_n^+\}$ and $N^- = \{v_1^-, \ldots, v_n^-\}$ the sets of pickup and delivery vertices, respectively. The sets N^+ and N^- are the sets of vertices of two complete, directed and unconnected graphs: The pickup graph $G^+(N^+, A^+)$, with $A^+ = \{(u^+, v^+), u^+, v^+ \in N^+\}$ and the delivery graph $G^-(N^-, A^-)$, with $A^- = \{(u^-, v^-), u^-, v^- \in N^-\}$. The vertices of the two graphs are connected to a depot, v_0, as shown in Figure 1a. Let $\{(v_0, v^+), (v^+, v_0), v^+ \in N^+\}$ and $\{(w^-, v_0), (v_0, w^-), w^- \in N^-\}$ be the sets of arcs that have v_0 as one of the extremes. A cost $c(u, v)$ is associated to each arc (u, v), where both u and v may be pickup vertices or delivery vertices, or one between u and v may be the depot and the other either a pickup or a delivery vertex.

A tour for the DTSP2S starts from the depot v_0, visits a sequence of n pickup vertices, returns to the depot, visits a sequence of n delivery vertices and finally returns again to the depot (Figure 1b). This means that the depot v_0 appears three times in a tour. In order to simplify the description of some important

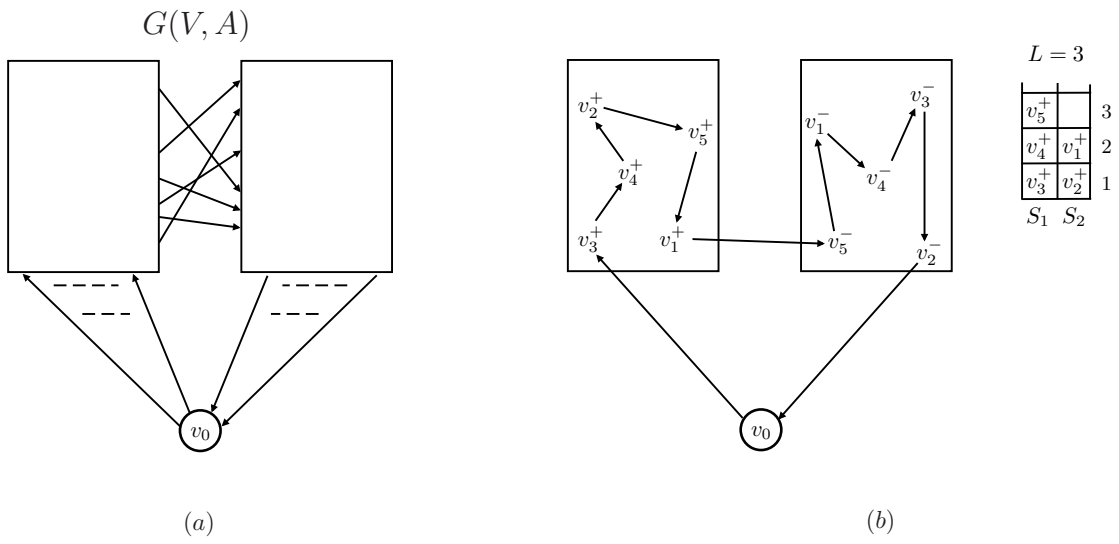


Figure 2: (a) The structure of the graph. (b) A feasible tour and a loading configuration associated to the pickup route $< v_3^+, v_4^+, v_2^+, v_5^+, v_1^+ >$.

aspects of the branch-and-bound approach we propose, such as the barrier vertex, the residual graph and the filters, it is convenient to work with the classical tour structure, i.e. a sequence of different vertices, starting and ending at the depot. To this end, we replace all the arc pairs (v^+, v^0), (v^0, w^-) with a single direct arc (v^+, w^-) where $c(v^+, w^-) = c(v^+, v^0) + c(v^0, w^-)$. We denote by $A^{\pm}$ this new set of arcs. In Figures 2a and 2b the new structure of the input graph and of a complete classical tour are shown, respectively. For each tour in the original graph, which in fact is composed by a pair of pickup and delivery tours, we have an equivalent tour with the same cost in the new graph and vice versa.

Hence, we will solve the DTSP2S on a directed and weighted graph $G = (V, A)$, where $V = \{\{v_0\} \cup N^+ \cup N^-\}$ and $A = \{\{(v_0, v^+)\} \cup A^+ \cup A^{\pm} \cup A^- \cup \{(v^-, v_0)\}\}$. We define a *pickup route* R^+ (*delivery route* R^-) as any sequence of n pickup (delivery) vertices. For instance, in Figure 2b we have $R^+ =< v_3^+, v_4^+, v_2^+, v_5^+, v_1^+ >$ and $R^- =< v_5^-, v_1^-, v_4^-, v_3^-, v_2^- >$.

The transportation is carried out using a single rear-loaded vehicle. The container of the vehicle is divided in 2 stacks $\{S_1, S_2\}$, each of given capacity L. We denote by $|S_i|$ the number of items in stack S_i. Any possible partial or complete loading of the two stacks is called a *loading configuration*. A loading configuration also includes the sequence of the items within each stack. The loading configuration $\{< v_3^+, v_4^+, v_5^+ >, < v_2^+, v_1^+ >\}$ at the end of the pickup tour is shown in Figure 2b. This is a *complete loading configuration* because all the pickup vertices have been visited and the pickup items have been accommodated into the stacks. With a little abuse of the language, we will sometimes say that a vertex is loaded in a stack.

The loading and unloading operations of each stack must be carried out in LIFO order, i.e. only the items visible from the rear of the container can be unloaded. We will use the representation of stacks as depicted in Figure 2b. Here, the items visible from the rear of the container are the items on top of the stacks.

Let R^+ and R^- be a pickup and a delivery route, respectively. These routes are feasible if and only if

the sequence of vertices in R^- satisfies the LIFO policy for the complete loading configuration associated with R^+. A tour T is feasible for DTSP2S if and only if the pickup and delivery routes contained in it are feasible. Our aim is to find the shortest feasible tour T^* in G.

We introduce now the definition of *consistent path* needed for the description of the search tree. A path p of G is called consistent if i) it starts from the depot v_0; ii) there exists a feasible tour T of G such that p is a subpath of the tour T; iii) it is associated with a loading configuration.

From the definition, it is easy to see that, in a consistent path, all precedence constraints are verified.

Given the tour T depicted in Figure 2b, any subpath starting from v_0 is a consistent path, when a loading configuration is associated to it. Notice that a stack has to be selected for each pickup vertex, where the picked up item is accommodated. Since, unless a stack is full, it is possible to choose between the two stacks, then different loading configurations may be associated with the same path. For instance, given the path $p_1 =< v_0, v_3^+, v_4^+, v_2^+ >$, possible loading configurations are $\{< v_3^+, v_4^+ >, < v_2^+ >\}$, $\{< v_3^+, v_4^+, v_2^+ >, < \emptyset >\}$, $\{< v_3^+ >, < v_4^+, v_2^+ >\}$, $\{< v_3^+, v_2^+ >, < v_4^+ >\}$. A consistent path p is characterized by two components: The sequence of vertices and a loading configuration. Two consistent paths are identical if both components coincide. Given a consistent path p, we define the following:

- $\mathcal{V}(p)$ is the set of vertices in p;
- $\mathscr{L}(p)$ is the last vertex of p;
- $\mathscr{C}(p)$ is the loading configuration associated with p.

Given the last vertex of p, $\mathscr{L}(p) = v$, we define the *residual graph* $G_v(p)$ as the subgraph of G induced by v and by the vertices that do not belong to p, i.e. $G_v(p) = (V_v(p), A_v(p))$ where $V_v(p) = \{V \setminus \mathcal{V}(p)\} \cup \{v\}$ and $A_v(p) = \{(x, y) : x, y \in V_v(p)\}$.

3. The Additive Branch-and-Bound Algorithm

In this section we briefly introduce the structure of the Additive Branch-and-Bound (ABB) algorithm and describe the various aspects that characterize it. Let $\hat{T}$ be the initial best solution computed by some heuristic. The search tree $\mathcal{T}$ built by ABB contains all the feasible tours of G. A vertex is added at each level of the search tree. The depot v_0 is added at level 0. From level 1 to level n the pickup vertices are added and from level $n+1$ to $2n$ the delivery vertices are added. The algorithm starts the construction of $\mathcal{T}$ from the depot v_0 and then extends any consistent path adding new vertices until a feasible tour is obtained. From level 0 to $n-1$ the branching strategy selects a new pickup vertex to add to the current path and a stack where to accommodate it. At the following levels it selects one of the delivery vertices whose pickup is on top of one of the two stacks. The exploration of the search tree is performed using the depth-first strategy. Before any branching operation, the ABB computes a lower bound on the residual graph induced by the current vertex and by the vertices that do not belong to the current path. To enforce the quality of lower bounds, we introduce a set of elimination rules (the *filters*) whose aim is to find and to remove from the residual graph the arcs that cannot belong to any feasible tour starting with the current path. Since the filtered graph contains less arcs than the original one, there are more chances to find tighter lower bounds. For the computation of the lower bounds, ABB applies an additive approach [13] based on two relaxations of the classical TSP: The assignment and arborescence problems. After the computation of the lower bounds, the algorithm checks whether the cost of the current path plus the best lower bound is lower than $c(\hat{T})$. If this is the case, the exploration continues, otherwise a pruning operation is performed. When the exploration reaches the level $2n$, a feasible tour T' is obtained and, if $c(T') < c(\hat{T})$, then $\hat{T}$ is updated with T'.

To improve the performance of the basic additive branch-and-bound algorithm, we use the following three tools: The barrier vertex, the polynomial computation of the delivery tours and the hashing strategy. Given a tour T, the barrier vertex z of T is the last pickup vertex. For any pickup vertex v_i^+, we build the subtree $\mathcal{T}_{v_i^+}$ containing all the paths starting from v_0 and having v_i^+ as barrier vertex z. We use a dynamic programming algorithm introduced in [5] that, given a complete loading configuration computes, in polynomial time, the optimal delivery tour that is compatible with this configuration. We implemented and embedded this algorithm into the ABB to complete the tour from the barrier vertex. The hashing strategy is used to identify redundant nodes of the search tree. As a result, more pruning operations are carried out in the search tree.

In the next parts of this section we describe in details the main features of ABB and in Section 4 we will show their impact on the performance of the algorithm.

3.1 Branching Strategy and Barrier Vertex

The ABB algorithm builds a search tree $\mathcal{T}$ containing all the feasible tours of G. A consistent path of G, denoted by $\rho(\tau)$, is associated with each node τ of $\mathcal{T}$. The loading configuration of the consistent path is $\mathscr{C}(\rho(\tau))$. Any feasible tour T of G is a consistent path with $2n+1$ vertices. At level $2n$ of $\mathcal{T}$ we will find all the feasible tours of G. We denote by $\ell(\tau)$ the level of a node τ in $\mathcal{T}$. To avoid confusion, from now on, we will use the term *vertex* to indicate one of the vertices of the graph G and the term *node* to indicate one of the elements of the search tree $\mathcal{T}$.

Since any consistent path starts from the depot, the root r of $\mathcal{T}$ corresponds to the trivial path containing v_0 only. This trivial path is extended on level 1 with one of the n pickup vertices, placing the corresponding item in S_1. No item is placed in S_2 on level 1 to avoid the creation of redundant nodes. Consequently, the branching of r generates n children, each one associated to a pickup vertex with its item placed in S_1 (Figure 3a).

After selecting a node τ on level 1, with $\mathscr{L}(\rho(\tau)) = v_1^+$, the consistent path $\rho(\tau) =< v_0, v_1^+ >$ can be extended with one of the remaining $n-1$ pickup vertices $\{v_2^+, \ldots, v_n^+\}$ and assigning the item to one of the two stacks (Figure 3b). In general, given any node $\tau \in \mathcal{T}$, the branching on it produces a set of nodes C_τ according to the following rules:

- if $\ell(\tau) = 0$, then, $\forall v^+ \in N^+ \, \exists \, \varphi \in C_\tau$;
- if $\ell(\tau) < n$, then, $\forall S_i$ that is not full and $\forall v^+$ outside $\rho(\tau), \exists \, \varphi \in C_\tau$;
- if $\ell(\tau) \geq n$, then, $\forall v^-$ outside $\rho(\tau)$ and having v^+ on top of a stack, $\exists \, \varphi \in C_\tau$.

The search tree $\mathcal{T}$ is divided in two parts. At the levels from 1 to n we find nodes associated with the pickup vertices while the following n levels are associated with the delivery vertices. It is important to notice here that the number of nodes created when branching at the early levels of the tree is considerably greater than the number of nodes generated from the delivery nodes that is equal to two, unless one of the stacks is empty.

We use the depth first policy to visit the tree. This policy was observed to be extremely efficient in terms of computational time.

The number of pruning operations performed on the search tree depends on the quality of the lower bounds computed on each node of $\mathcal{T}$. Since the nodes are largely located on the first $n+1$ levels, we focus the attention on how to compute the lower bounds at these levels. To this aim, we introduce the concept of *barrier vertex* that changes the construction of the search tree to reduce the number of nodes at the first $n+1$ levels.

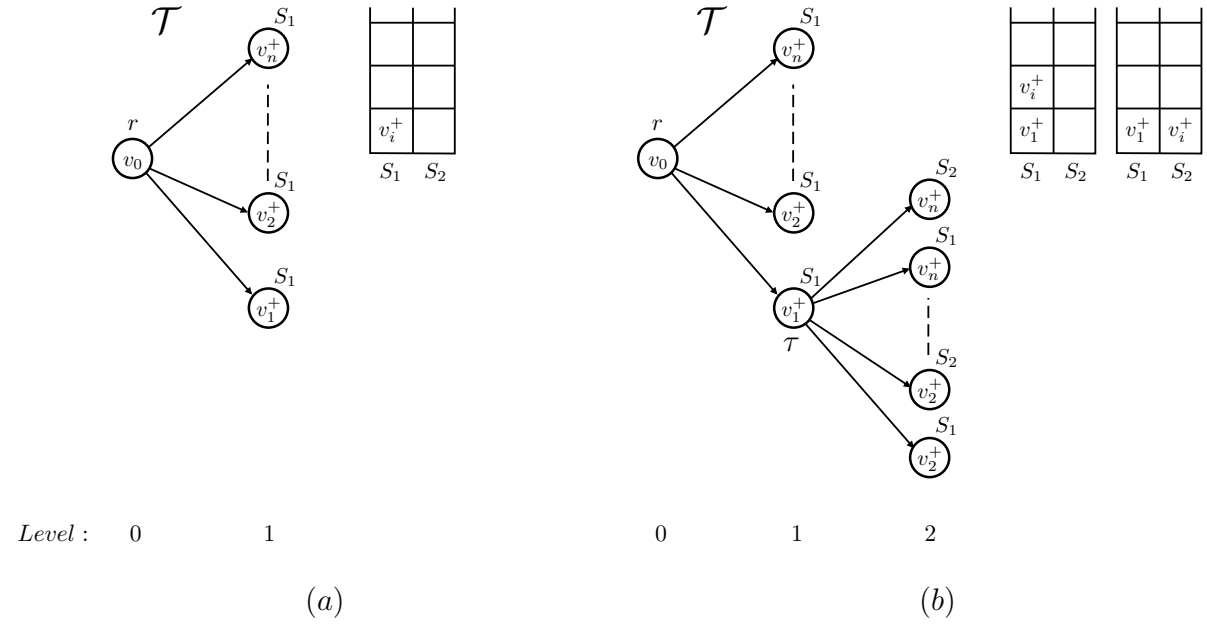


Figure 3: The branching strategy. (a) The branching on the root r generates n nodes. (b) The branching on node τ, associated with pickup vertex v_1^+, generates $2(n-1)$ children, associated with the remaining $(n-1)$ pickup vertices placed in S_1 and S_2.

By construction, while $|A^{\pm}| = n^2$, obviously any feasible tour T of G contains only one arc $(v^+, w^-) \in A^{\pm}$. We focus the attention on the last pickup vertex that we define to be the barrier vertex z. We first select a pickup vertex as possible barrier vertex z and later build all the feasible tours having z at level n. Obviously, the selection of the barrier vertex is carried out n times, one for each pickup vertex because the barrier vertex inside the optimal tour is unknown. After the barrier vertex z has been chosen at the root node, the algorithm proceeds with the normal branching, using the remaining $n-1$ pickup vertices, to build the subtree $\mathcal{T}_z$ containing all the feasible tours that have z at level n. Therefore, we can divide the new search tree $\mathcal{T}$ in n subtrees $\mathcal{T}_{v_1^+}, \ldots, \mathcal{T}_{v_n^+}$ each one rooted in r and having v_i^+, with $1 \leq i \leq n$, as barrier vertex (Figure 4).

We have two important advantages derived from the introduction of the barrier vertex. The first is the identification of new filters, described in Section 3.3, that improve the lower bounds. The second advantage concerns the introduction of a stopping criterion that avoids the complete exploration of the subtree $\mathcal{T}_{v_i^+}$. To this end, the sequence of barrier vertices is carefully generated. For each $v_i^+ \in N^+$, we compute the cheapest Hamiltonian tour $T^*_{v_i^+}$ of G where v_i^+ is the last pickup vertex. This can be done by setting $c(v^+, w^-) = \infty$, $\forall v^+ \in N^+ \setminus \{v_i^+\}, \forall w^- \in N^-$, in the cost matrix of G and then solving the classical TSP on this modified matrix. The resulting n tours are sorted according to their non-decreasing cost and the subtrees $\mathcal{T}_{v_i^+}$ will be explored according to this order. Obviously, $c(T^*_{v_i^+})$ is a lower bound to the value of any feasible tour in $\mathcal{T}_{v_i^+}$. For this reason if, during the exploration of this subtree, the algorithm finds a feasible tour T with $c(T) = c(T^*_{v_i^+})$, then the algorithm stops, returning the optimal tour. Indeed, any feasible tour

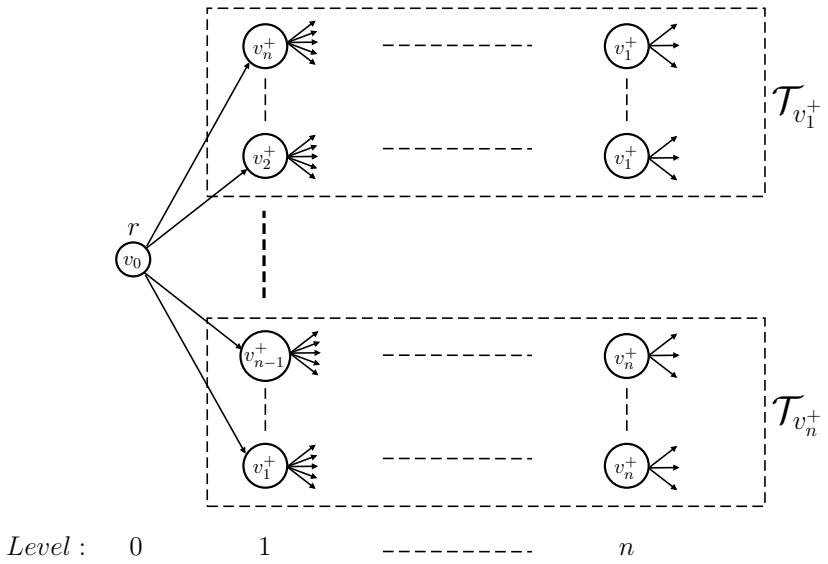


Figure 4: The new search tree after the introduction of the barrier vertex.

in the remaining subtrees will have a cost at least equal to $c(T)$. Moreover, before starting the exploration of any subtree $\mathcal{T}_{v_i^+}$, the algorithm checks whether $c(T^*_{v_i^+})$ is lower than the cost of the incumbent solution. If this is the case, the exploration starts, otherwise the algorithm stops.

A drawback of the introduction of the barrier vertex is the duplication of nodes on levels from 1 to n of the search tree. Let us consider Figure 5a in which the first $n = 3$ levels of the search tree $\mathcal{T}'$ generated by the original branching strategy are shown. For sake of simplicity, we do not consider the stacks here. The total number of nodes of $\mathcal{T}'$ is equal to 16. In Figure 5b the search tree $\mathcal{T}''$ built using the barrier vertex is shown. Each node v_i, with $1 \leq i \leq 3$, appears two times on the level 1 of $\mathcal{T}''$ while in $\mathcal{T}'$ only once and $\mathcal{T}''$ contains three nodes more than $\mathcal{T}'$. This problem worsens when introducing the stacks and adding other vertices. Despite that, the test results reveal that the improvements introduced by the barrier vertex overcome this drawback. This happens essentially because many of the duplicated vertices are quickly pruned on the first levels of the search tree, thanks to the tighter lower bounds derived by the use of the barrier vertex. Moreover, for each node τ, with $\ell(\tau) < n - 1$, the set of children C_τ cannot contain the barrier vertex. This implies that the size of each subtree $\mathcal{T}_{v_i^+}$ is much smaller than that of the original tree. To see that, let us consider again Figure 4 and let v_1^+ be the current barrier vertex. Then, at level 1 of $\mathcal{T}_{v_1^+}$ we will have $n - 1$ nodes associated with the vertices $v_2^+, \ldots, v_n^+$. At level 2, since there are two stacks, we have two nodes less for each node of level 1, that is on level 2 there are $(n-1) \times 2(n-2)$ nodes while in the original tree there was $n \times 2(n-1)$.

3.2 Lower Bound Computation

We describe here the computation of the lower bounds performed at each node of the search tree. This is a crucial aspect because the performance of ABB depends on the size of the search tree and the pruning

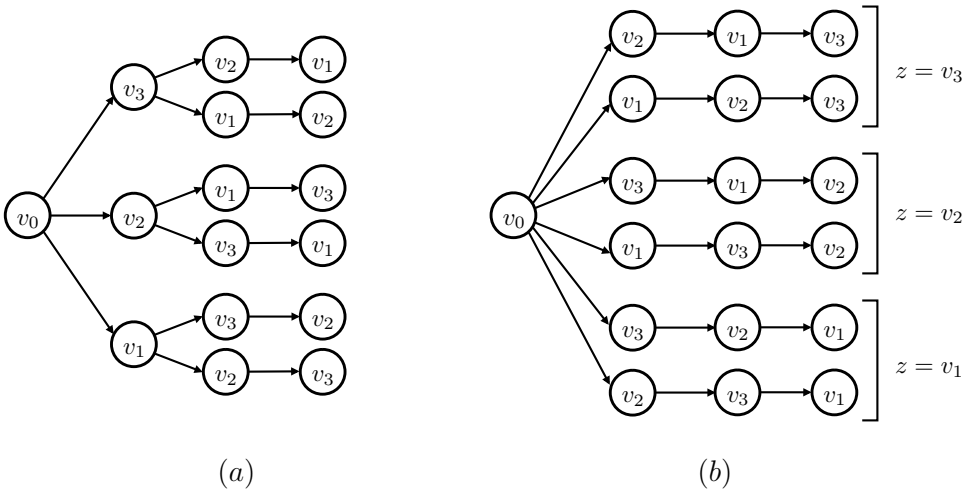


Figure 5: Redundance produced by barrier vertex.

operations reduce this size.

Given a node $\tau \in \mathcal{T}$ with $\mathscr{L}(\rho(\tau)) \neq v_0$, let $\hat{T}$ be the best tour found so far and G_τ be the residual graph induced by $\rho(\tau)$. In order to build a feasible tour T, we have to find a path $\overline{\rho}(\tau)$ from $\mathscr{L}(\rho(\tau))$ to v_0 such that $\rho(\tau)$ and $\overline{\rho}(\tau)$ are together a feasible tour of DTSP2S. A path $\overline{\rho}(\tau)$ is called a *residual path* of G_τ. Now, let lb_τ be a lower bound to the cost of all residual paths in G_τ. If $c(\rho(\tau)) + lb_\tau \geq c(\hat{T})$, then it is useless to continue the exploration of the subtree rooted in τ because it does not contain feasible tours better than $\hat{T}$. In this situation the algorithm prunes τ and backtracks to its father. The computation of the lower bounds is performed using two classical relaxations of the TSP: The *assignment problem* (AP) and the *r-shortest spanning arborescence problem* (r-SAP), respectively. The first problem can be solved in $O(|V|^3)$ using the Hungarian algorithm [1], while the second one in $O(|A|log|V|)$ using the efficient implementation of Edmonds' algorithm [10] proposed by Tarjan [23]. Unfortunately, the lower bounds computed by these two relaxations are good for the classical TSP but poor for the DTSP2S.

In order to obtain an effective lower bound, we apply an additive approach that uses the two relaxations simultaneously. This approach can be outlined as follows in the context of the TSP (for a detailed description we refer to [13, 14]). Let $\mathscr{B}^{(1)}, \mathscr{B}^{(2)}, \ldots, \mathscr{B}^{(q)}$ be q bounding procedures available for the TSP. The application of one of these procedures, say $\mathscr{B}^{(h)}$, on the cost matrix $c = (c_{ij})$ produces a lower bound $l^{(h)}$ and a *residual cost* matrix $c^{(h)}$ such that:

- $c_{ij}^{(h)} \geq 0$ for all $i, j \in V$;
- $l^{(h)} + c^{(h)}(T) \leq c(T)$ for any feasible tour T. In other words, the sum of the lower bound and the cost of T on $c^{(h)}$, is not greater than the cost of T on c.

Given a node $\tau \in \mathcal{T}$ and the residual graph G_τ induced by $\rho(\tau)$, we solve the assignment problem computing a first lower bound l^1 and a residual cost matrix c^1. Then on this matrix c^1 the r-SAP problem is solved obtaining another lower bound l^2. From the properties of the residual cost matrix mentioned above, $l^1 + l^2$ is a lower bound to the cost of all residual paths in G_τ. A more detailed description of our implementation can be found in [3].

3.3 Filters

In this section we show how to improve the lower bounds generated by the additive approach by using the specific structure and constraints of the problem. During the construction of a consistent path $\rho(\tau)$, precedence constraints among the vertices loaded in the same stack are generated. Because of these constraints, several arcs of the residual graph G_τ cannot belong to any residual path $\overline{\rho}(\tau)$. However, if these arcs are selected in the solutions produced by the AP and the r-SAP relaxations, the quality of the lower bounds worsens. To avoid this problem, we introduce a set of elimination rules, called *filters*, whose aim is to remove as many arcs as possible from the residual graph before computing the lower bounds. Since the filtered residual graph contains a reduced set of arcs with respect to the original one, the relaxations will in general produce better lower bounds.

Before describing the filters, we introduce some definitions. Given a consistent path p and a vertex v, we denote by $pos(v)$ the position of v in p. In Figure 2b, for example, $pos(v_3^+) = 1$ and $pos(v_3^-) = 9$. We define $pos(v_0) = 0$. Given two vertices v and w in p, if $pos(v) < pos(w)$, then v *precedes* w in the path. Obviously, $pos(v^+) < pos(w^-)$ $\forall v^+ \in N^+, w^- \in N^-$. We denote by $\mathcal{S}(v^+)$ and $\mathcal{P}(v^+)$ the stack where v^+ is loaded and the position of v^+ in the stack, respectively. If v^+ is not loaded in any stack then the stack and the position take value -1.

We are now ready to describe the seven filters applied in our branch-and-bound algorithm. Here p denotes the consistent path we are building, with z as barrier vertex.

*f*1. Remove (v^+, w^-), and (z, v^+), $\forall v^+ \in N^+ \setminus \{z\}, w^- \in N^-$, before starting the construction of p.
Since the barrier vertex z is the last pickup vertex in p, all the arcs from a pickup vertex, different from z, to a delivery vertex can be removed. Moreover, also the arcs from z to the other pickup vertices can be removed because no pickup vertex can follow z in p.

*f*2. If $\mathscr{L}(p) = v^+$ and $pos(v^+) < n - 1$, then remove (v^+, z).
Since $pos(z) = n$, any pickup vertex v^+ located in a position lower than $n - 1$ cannot be inserted immediately before z. Therefore, the arc (v^+, z) can be removed.

*f*3. If $\mathscr{L}(p) = v^+$ and another vertex will be added in $\mathcal{S}(v^+)$ after v^+, then the arc (z, v^-) can be removed.
Because of the $LIFO$ constraints, from z we can reach one of delivery vertices whose pickup vertex is on top of some stack. Since v^+ will not be on top of $\mathcal{S}(v^+)$ when z is reached, then (z, v^-) can be removed. To verify whether other vertices will be added to the same stack of v^+, we check whether there is enough space, inside the other stack, to load the pickup vertices in $N^+ \setminus \mathcal{V}(p)$.

*f*4. If there are no empty stacks, then remove (v^-, v_0), $\forall v^+ \in N^+ : \mathcal{P}(v^+) > 1$ or $\mathcal{P}(v^+) = -1$.
A delivery vertex w^- can be inserted immediately before the depot only if $\mathcal{P}(w^+) = 1$. Therefore, once the first position of both stacks has been filled, then it is possible to remove all the arcs to v_0 except for the arcs coming from a delivery vertex whose pickup vertex is in position 1 of some stack. Notice that this filter removes also arcs originating at delivery vertices whose pickup is not yet loaded.

*f*5. If $\mathscr{L}(p) = v^+$ and $\mathcal{P}(v^+) > 1$, then remove (v^-, v_0).
The difference with respect to the previous filter is that this one is applied also with empty stacks.

*f*6. If $\mathcal{S}(v^+) = \mathcal{S}(w^+)$ and $\mathcal{P}(v^+) < \mathcal{P}(w^+)$, then remove (v^-, w^-).
Since v^+ is under w^+ in the stack, then w^- has to precede v^- in p (Figure 6a). Hence the arc (v^-, w^-) can be removed.

*f*7. If $\mathcal{S}(v^+) = \mathcal{S}(w^+)$ and $\mathcal{P}(v^+) = \mathcal{P}(w^+) - 1$, then remove (z, v^-) and (w^-, u^-), $\forall u^-$ such that $\mathcal{S}(u^+) = \mathcal{S}(v^+)$ and $\mathcal{P}(u^+) < \mathcal{P}(v^+)$.

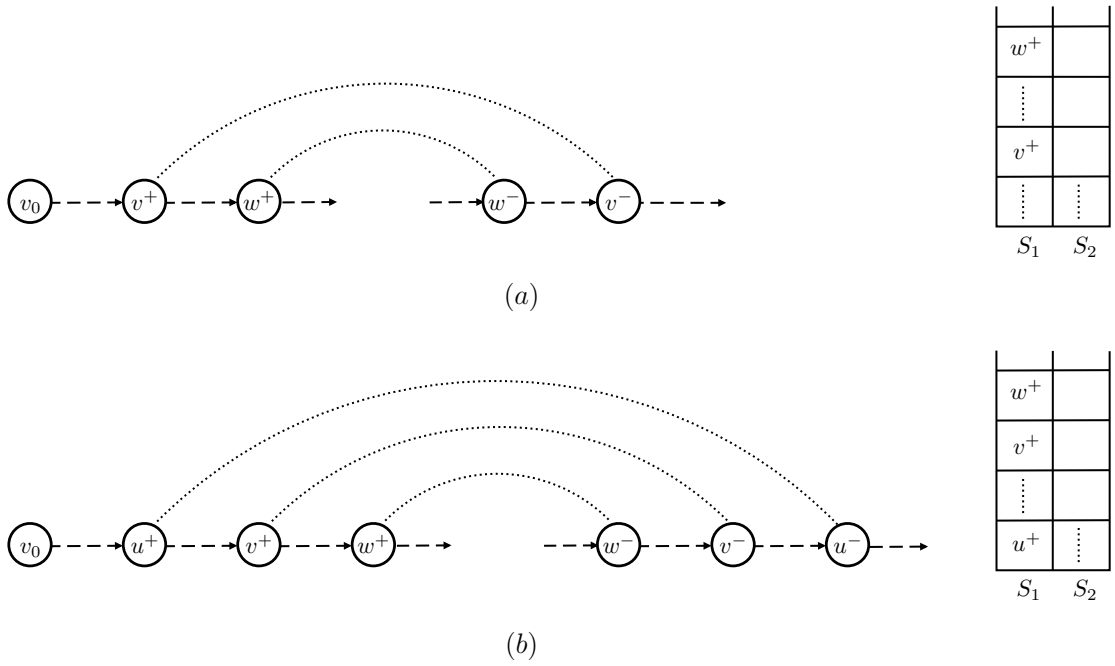


Figure 6: Graphical representation of filters $f6$ and $f7$.

The loading constraints impose that w^- is visited before v^- and then v^- cannot be visited immediately after z. Moreover, v^- has to precede any other delivery vertex whose pickup vertex is under v^+ in $\mathcal{S}(v^+)$ (Figure 6b) and then any arc from w^- to one of these delivery vertices can be removed.

From now on, we will consider the residual graph G_τ obtained after the application of the filters. The filters $f1$ and $f2$ are generated thanks to the introduction of the barrier vertex and they are the only filters that are able to remove arcs from a pickup vertex to another pickup vertex. Notice that the filters are only applied to consistent paths ending with a pickup vertex. This is due to the fact that, thanks to the dynamic programming algorithm described in Section 3.4, we avoid the exploration of the levels of the search tree associated with delivery vertices and then the construction of consistent paths stops at the barrier vertex.

3.4 Polynomial Computation of Delivery Tours

When a barrier vertex is reached in the search tree, the optimal delivery tour is calculated with the polynomial algorithm presented in [5]. The authors showed that, given a complete loading configuration, it is possible to compute in polynomial time a delivery tour which is feasible with respect to the loading configuration and has minimum cost.

We briefly describe here the dynamic programming algorithm, *DPA*, that runs in polynomial time and computes the optimal delivery tour. The construction of the delivery tour is carried out by adding at the end of a partial tour one of the delivery vertices whose pickup is on top of a stack. Let us define $f(x_1, x_2, i)$ the value of the shortest path starting from v_0, having x_1 items in the stack S_1, x_2 items in the stack S_2,

and whose next vertex is the delivery vertex v^- whose pickup vertex is on top of S_i. If v^- is the first delivery vertex of the tour then $f(x_1, x_2, i) = c(v_0, v^-)$. Otherwise, $f(x_1, x_2, i) = \min\{f(x_1 + 1, x_2, 1) + c(w_1^-, v^-), f(x_1, x_2 + 1, 2) + c(w_2^-, v^-)\}$, where w_j^- is the delivery vertex whose pickup vertex is on top of S_j. Using this dynamic programming recursion, it is possible to compute all the values of $f()$, and then the best delivery tour, in $O(n^3)$.

After the computation of the best delivery tour, it is sufficient to add $c(z, v_0)$ to its cost to obtain the value of the best residual path starting from z. Notice that setting $f(x_1, x_2, i) = c(z, v^-)$, for the base case of recursion, the DPA returns directly the value of the best residual path. However, we do not proceed in this way because, as we will see in the next section, we apply the hashing strategy to DPA.

Thanks to the DPA, the branch-and-bound stops the construction of the search tree at the barrier vertices avoiding the exploration of the subtrees rooted in them. Indeed, every time the ABB reaches the level n, it invokes DPA to compute the best delivery tour and then the residual path. To evaluate the impact of the dynamic programming algorithm for the calculation of the delivery tours, we also implemented the complete exploration of the search tree including the explicit exploration of the nodes associated with delivery vertices. The dynamic programming algorithm reduces the computational time of the complete algorithm but not as much as we expected. This is probably due to the fact that the second half of the search tree contains much fewer nodes (all the subtrees rooted into the barrier vertices are binary trees) than the first half. Moreover, most of them are pruned because the lower bounds are tighter on these levels. In the next section we will describe the real advantage derived from the application of DPA.

3.5 Hashing Strategy

In this section we introduce an important aspect of our branch-and-bound approach, the hashing strategy.

During the exploration of the search tree, important information, such as the lower bounds and the cost of the consistent path associated with each node, is computed and later lost. This is a normal situation in a classical branch-and-bound approach because the information cannot be reused. This is not the case for the DTSP2S problem. By analyzing the search tree, we discovered that there are several nodes where it is possible to reuse this information to quickly decide if they can be pruned because no better solution can be found in the subtree rooted in them. The nodes on which it is possible to reuse the information are defined *redundant nodes*.

In order to identify the redundant nodes, we use the concept of equivalent paths. Given a loading configuration, different consistent paths may share this loading configuration. The subset of these paths that end with the same vertex are called *equivalent*. Formally:

Definition 1. Consistent paths $p_1, p_2 \ldots, p_t$ are *equivalent* if and only if $\mathscr{C}(p_1) = \mathscr{C}(p_2) = \ldots = \mathscr{C}(p_t)$ and $\mathscr{L}(p_1) = \mathscr{L}(p_2) = \ldots \mathscr{L}(p_t)$.

For instance, the loading configuration $\{< v_3^+, v_4^+, v_2^+ >, < v_1^+, v_5^+ >\}$ depicted in Figure 7a is generated by the six equivalent paths shown in Figure 7b. Once one of these equivalent paths has been built, the nodes associated with the remaining five paths become redundant.

The equivalent paths share an important property.

Lemma 1. *Let z be a barrier vertex and p_1 and p_2 two equivalent paths with $\mathscr{L}(p_1) = v^+$. Then, the filters remove exactly the same arcs from the residual graphs induced by p_1 and p_2.*

Proof. The only difference between p_1 and p_2 is the sequence of the vertices that belong to them. Whereas, during the construction of the two paths, the loading configurations may be different, they coincide when the two paths reach v^+. To prove that the set of arcs of the two residual graphs induced by p_1 and

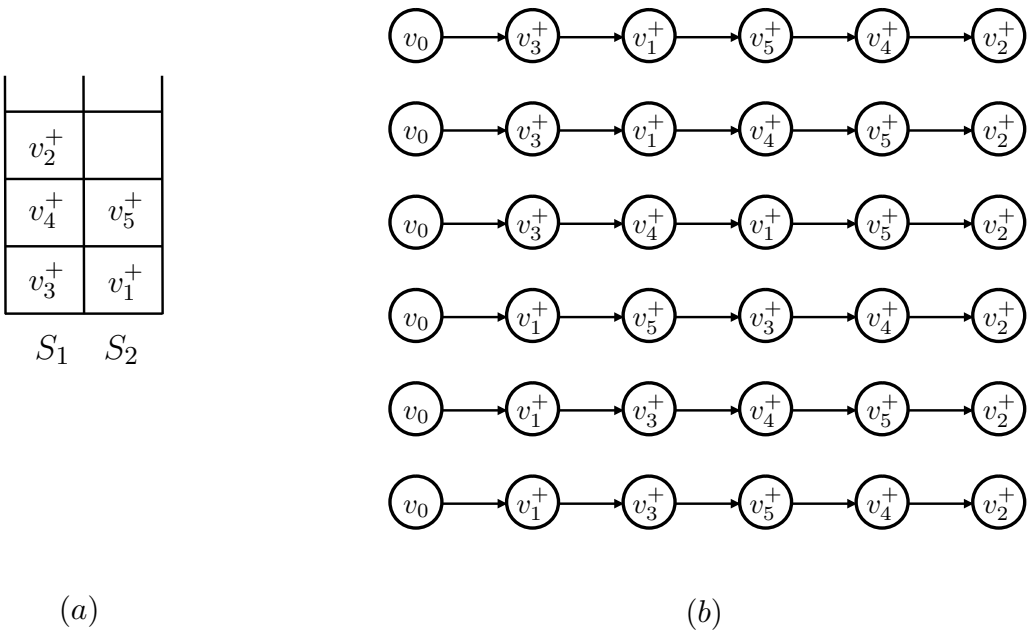


Figure 7: (a) A loading configuration and (b) the equivalent paths derivable from it.

p_2 coincide, we analyze the behavior of the filters. Let us start from the first filter $f1$. This filter is applied immediately after the selection of the barrier vertex z and before starting the construction of any consistent path with this barrier vertex. Since the arcs removed depend only on z, they are the same for p_1 and p_2. The filter $f2$ removes only one arc outgoing from the last vertex of the path. Since p_1 and p_2 end with the same vertex v^+, if the arc (v^+, z) satisfies the condition of $f2$, it is removed in both cases. The proof for $f3$ is a bit more complicated. Let u^+ be a vertex belonging to p_1 and p_2 and w.l.o.g. let us assume that it is located in two different positions in these two paths. Moreover, let p'_1 and p'_2 be the subpaths of p_1 and p_2 ending with u^+. Obviously, $\mathcal{S}(u^+) = S_1$ is the same for p'_1 and p'_2, while the other stack S_2 can be loaded in different way. After p'_1 has been built, in order to check whether u^+ will be covered in the stack, the filter $f3$ verifies the following condition: $L - |S_2| < |V \setminus \mathcal{V}(p'_1)|$. If this condition holds, then there is not enough space in S_2 to contain all the pickup vertices outside p'_1 and then at least another vertex will be inserted in S_1 after u^+. It is easy to see that for each vertex added in S_2 both members of the inequalities are decreased by one while for each vertex removed from S_2 both members are increased by one. This means that the loading of S_2 cannot change the result of the condition and then if the condition holds for u^+ in p'_1 then it holds also for u^+ in p'_2. The case when the condition does not hold is proven in the same way. About the filter $f4$, since $\mathscr{C}(p_1) = \mathscr{C}(p_2)$, whatever the position of a vertex u^+ is in the two paths, if $\mathcal{P}(u^+) > 1$ or $\mathcal{P}(u^+) = -1$ in p_1, the same holds also in p_2 and vice versa. During the construction of p_1 and p_2, only the iteration when the arc (v^-, v_0) is removed could be different, because it depends on the position of u^+ in these paths. A similar reasoning holds also for $f5$, $f6$ and $f7$. □

Lemma 2. *Let z be a barrier vertex and $p_1, \ldots, p_t$ a set of equivalent paths in $\mathcal{T}_z$. Then the residual graphs induced by these paths coincide.*

Proof. Since $p_1, \ldots, p_t$ contain the same vertices and $\mathscr{L}(p_1) = \ldots = \mathscr{L}(p_t) = v$, by definition, the vertex set V_v of the induced residual graphs is the same. It remains to be proved that also the set of arcs A_v is the same. Obviously, without the application of the filters we have that $A_v = \{(u, v) : u, v \in V_v\}$ for all the residual graphs. Therefore, only the application of the filters could generate difference sets of arcs. However, this is not possible, due to Lemma 1. □

From the above Lemma, we can derive the following important result.

Theorem 1. *Let z be a barrier vertex and $p_1, \ldots, p_t$ a set of equivalent consistent paths in $\mathcal{T}_z$. If $\tau_1, \ldots, \tau_t$ are the nodes of the search tree associated with these consistent paths, then $lb_{\tau_1} = \ldots = lb_{\tau_t}$.*

Proof. Since, from Lemma 2, $p_1, \ldots, p_t$ induce the same residual graph, also the lower bounds computed on the residual graphs coincide. □

Since the computation of the lower bounds is a computationally intensive operation, the above result has a relevant computational impact. To take advantage of this result, we need a data structure where to save the value of the lower bound associated with a consistent path p_1 in such a way that when a new equivalent path p_2 is built, we can retrieve this value. To this end we use a hash table $\mathcal{H}$ and define the function $\kappa(p)$ that, given a consistent path p, generates the hash key using $\mathscr{C}(p)$ and $\mathscr{L}(p)$. Obviously, p_1 and p_2 are equivalent if and only if $\kappa(p_1) = \kappa(p_2)$. Let us see, in more details, how the hash key is generated. For each stack S_i, we generate a string containing the vertices of the stack from the bottom to the top. Then, we sort the strings obtained according to the index of the first vertex of the string and concatenate them. Finally, we add to the end of the complete string just built the vertex $\mathscr{L}(p)$. For instance, the hash key for the configuration shown in Figure 7a is: $v_1^+, v_5^+/v_3^+, v_4^+, v_2^+|v_2^+$. Notice that, thanks to the strings sorting, swapping the content of two stacks generates the same hash key.

The advantages of the hashing strategy are not limited to the saving of the lower bound computations. As described at the beginning of this section, we aim at pruning as many redundant nodes as possible. Therefore, we use the cost of the path associated with the redundant node and Lemma 2. Given the barrier vertex z, let $\rho(\tau_1)$ and $\rho(\tau_2)$ be two equivalent consistent paths. From Lemma 2, the residual graph associated with the two paths is the same and then a feasible solution $T_2 = \rho(\tau_2) \cdot \overline{\rho}(\tau_2)$ with $\overline{\rho}(\tau_1) = \overline{\rho}(\tau_2)$ corresponds to each feasible solution $T_1 = \rho(\tau_1) \cdot \overline{\rho}(\tau_1)$ and vice versa. Since the residual paths coincide, the different values of $c(T_1)$ and $c(T_2)$ depend on $c(\rho(\tau_1))$ and $c(\rho(\tau_2))$ only. W.l.o.g., if $c(\rho(\tau_1)) \leq c(\rho(\tau_2))$ then the cost of any feasible solution containing $\rho(\tau_2)$ is at least equal to the cost of the corresponding solution containing $\rho(\tau_1)$. Therefore, if the algorithm visits the node τ_1 first, the exploration of the subtree rooted in τ_2 becomes useless and then τ_2 can be pruned. Vice versa, if τ_2 is visited first, then the algorithm has to explore also the subtree rooted in τ_1. However, this happens only if $c(\rho(\tau_1))$, summed to lb_{τ_1}, is lower than $c(\hat{T})$. To carry out the pruning of redundant nodes we need to know the cost of any consistent path p_1 whose lower bound is saved in $\mathcal{H}$. To this goal, we save two data in the hash table: The cost of the path and the value of the lower bound computed on the residual graph. This information represents an *element* of the hash table.

We illustrated with an example the use of the hash table. Given a hash key $\kappa(p_1)$, let $\alpha(\kappa(p_1))$ and $\beta(\kappa(p_1))$ be the cost and the lower bound saved in the cell $\kappa(p_1)$ of $\mathcal{H}$, respectively. Given the barrier vertex z, let us suppose that in $\mathcal{T}_z$ the algorithm builds the equivalent paths $p_1, \ldots, p_6$ depicted in Figure 8a, where p_1 is built before p_2 and so on. The cost of each path is shown on the right of the path. Moreover, let $\hat{T}$ be the best solution found so far with $c(\hat{T}) = 164$. When the algorithm builds p_1, there is no information saved in the cell $\kappa(p_1)$. For this reason, the lower bound lb is computed on the residual graph induced by p_1 and saved together with $c(p_1)$ in $\kappa(p_1)$ (Figure 8b). Then, the algorithm checks whether the cost of the current path summed to the just computed lower bound is lower than the cost of current best solution. Since in this case $c(p_1) + lb > c(\hat{T})$, then the current node is pruned. After some iterations, ABB builds the path p_2, that

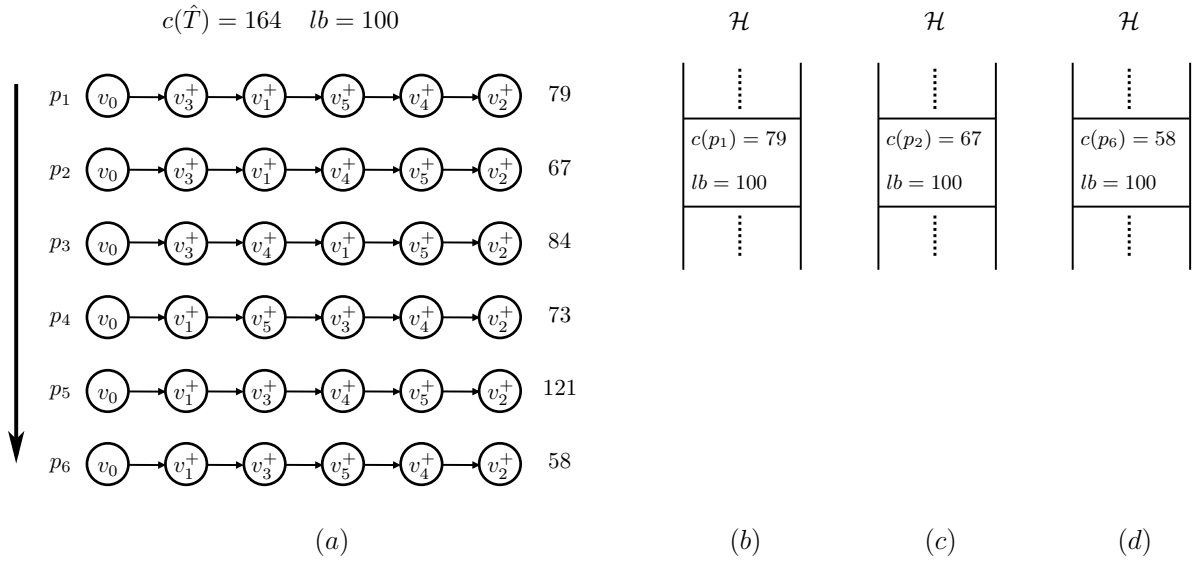


Figure 8: The hash table

is equivalent to p_1. Since $\kappa(p_2) = \kappa(p_1)$, the associated cell in the hash table is not empty and then the value of the lower bound can be directly retrieved from $\kappa(p_2)$. Since $c(p_2) < \alpha(\kappa(p_2)) = 79$, then the new value is saved in the cell (Figure 8c). However, also in this case the current node is pruned because $c(p_2) + lb > c(\hat{T})$. When p_3 is built, the algorithm immediately prunes the associated node because $c(p_3) > \alpha(\kappa(p_3)) = 67$. This is done also for p_4 and p_5. Finally, when p_6 is built, since $c(p_6) < \alpha(\kappa(p_6))$, the new value is saved in the hash table (Figure 8d). Moreover, since the condition $c(p_6) + lb < c(\hat{T})$ holds, the subtree is explored.

As it should be clear from the above description, the number of elements inserted in the hash table is related to the number of loading configurations generated by different consistent paths of the search tree. Since this number grows exponentially, it is impossible to save all of them in the hash table. To face this problem, we select some levels of the search tree and save in the hash table the information associated with the nodes on these levels only. The selection of the levels is based on the probability to carry out pruning operations on them. For instance, let $p_1 =< v_0, v_1^+, v_2^+, v_3^+ >$ be a consistent path with $\mathscr{C}(p_1) = \{< v_1^+, v_3^+ >, < v_2^+ >\}$ and let τ be the node of the search tree associated with this path. Obviously, τ is located on level 3 of $\mathcal{T}$. It is easy to see that only another consistent path $p_2 =< v_0, v_2^+, v_1^+, v_3^+ >$ exists which is equivalent to p_1. Therefore, at most one pruning operation can be performed by saving in the hash table the information associated with p_1. For this reason, it is unattractive to select level 3 of the search tree. Obviously, whereas the probability to carry out pruning operations is greater at earlier levels of the search tree, also the number of elements to be saved in the hash table increases at those levels. On the other hand, a pruning operation performed at later levels has a greater impact on the performance of the algorithm. The best situation occurs when there is enough space in the hash table to save all the elements. However, when this is not possible it is necessary to find a tradeoff between the quantity and quality of the selected levels and the limited available memory. Here, the importance of DPA becomes more evident. Without this algorithm, the hash table should handle $2n$ levels while with this algorithm the number of levels is reduced to n. As a consequence, more elements of the first n levels can be saved and the number of pruning operations

performed on the first n levels of $\mathcal{T}$ increases.

Another interesting use of the hash table concerns the level n. As explained above, the elements of $\mathcal{H}$ contain the cost of the path and the lower bound computed on the residual graph. Since on the level n the DPA is invoked, by changing the hash key generation (as we will see below, $\mathscr{L}(p)$ here is useless) and the kind of elements to store, it is possible to reduce the total number of DPA invocations. Let us see how.

Let p_1 and p_2 be two equivalent paths containing all the pickup vertices and w.l.o.g. let us suppose that p_1 is built before p_2 in $\mathcal{T}$. Since $\mathscr{C}(p_1) = \mathscr{C}(p_2)$, the DPA returns the same delivery tour when invoked on $\mathscr{L}(p_1)$ and $\mathscr{L}(p_2)$. Therefore, if we save the cost of this delivery tour then, when p_2 is built, we can directly retrieve from the hash table its cost instead of invoking again DPA to compute the same tour. We denote by $\mathcal{H}_n$ the hash table used at level n. Its elements are composed of a single information: the value of the best delivery tour associated with the given loading configuration. Since the solution computed by DPA depends only on the loading configuration $\mathscr{C}(p)$ associated with the consistent path p, the hash key for $\mathcal{H}_n$ is generated using this information only ($\mathscr{L}(p)$ here is useless).

So far we have described the hashing strategy, given a fixed barrier vertex z. It is interesting now to see the behavior of ABB algorithm when the barrier vertex changes. Let us start from the hash table $\mathcal{H}$. All the lower bounds on the nodes of $\mathcal{T}_z$ are computed by forcing z to be the last pickup vertex. Consequently, when the barrier vertex changes, the lower bounds previously computed do not hold. On the contrary, the cost of the consistent paths, which is saved in the hash table, always holds whatever the barrier vertex is. This cost can be reused to carry out pruning operations. Indeed, let us suppose that $c(p_1)$ is the cost of a consistent path in $\mathcal{T}_z$, saved in $\mathcal{H}$, and let p_2 be an equivalent path in $\mathcal{T}_{z'}$ built later. If $c(p_2) \geq c(p_1) + 1$, then we can prune the node associated with p_2 because we know that there is a cheaper consistent path equivalent to p_2 whose cost is equal to $c(p_1)$. Instead, if $c(p_2) < c(p_1) + 1$, then the algorithm updates the value in the hash table and continues the exploration. From previous observations, every time the barrier vertex changes, ABB performs one of following two operations: it empties the hash table, if it is full, or it sets to 0 the lower bounds saved in $\mathcal{H}$ and increases by one the cost of the consistent paths. We need to increase this cost by one to avoid a incorrect pruning operation when the cost corresponds with the cost of the cheapest consistent path.

It is interesting to note the consequence of the change of the barrier vertex on $\mathcal{H}_n$. Let p_1 and p_2 be two paths containing all the pickup vertices and such that $\mathscr{C}(p_1) = \mathscr{C}(p_2)$ while $\mathscr{L}(p_1) \neq \mathscr{L}(p_2)$ because we changed the barrier vertex. W.l.o.g. let us suppose that p_1 is built before p_2 in $\mathcal{T}$. Since $\mathscr{C}(p_1) = \mathscr{C}(p_2)$, the DPA returns the same delivery tour when invoked on $\mathscr{L}(p_1)$ and $\mathscr{L}(p_2)$. Therefore, the possibly different cost of residual paths starting from $\mathscr{L}(p_1)$ and $\mathscr{L}(p_2)$ depends only on $c(\mathscr{L}(p_1), v_0)$ and $c(\mathscr{L}(p_2), v_0)$. This means that we can reuse the costs stored in $\mathcal{H}_n$ also after changing the barrier vertex. For this reason, no operations are performed on this table when the barrier vertex changes.

4. Computational Results

The ABB algorithm was coded in C and run on a 2.33 GHz Intel Core2 Q8200 processor. Following both Lusby et al. [20] and Petersen et al. [21], the ABB algorithm was tested on a set of benchmark instances used in [22] and available at: *http://www.imm.dtu.dk/∼hlp/data/*. These data sets have 33 customers (i.e. 66 vertices plus the depot) randomly located on a 100×100 grid. The depot is located at the point (50.0, 50.0). The cost of the arcs is the Euclidean distance rounded to the nearest integer. All the tests were carried out with a capacity of each stack equal to $\lceil n/2 \rceil$. A maximum CPU time of 1 hour was imposed for the solution of each instance. An initial upper bound for the ABB algorithm is computed using the large neighborhood search (LNS) proposed by Côté et al. [8]. This heuristic is run 10 times. The initial upper bound is the best value increased by one. In this way, when the LNS finds the optimal solution, also the ABB algorithm can

Size →	25						27						29					
Inst. ↓	**UB**	**Best**	**Ndpa**	**Nbar**	**Nhash**	**ABB**	**UB**	**Best**	**Ndpa**	**Nbar**	**Nhash**	**ABB**	**UB**	**Best**	**Ndpa**	**Nbar**	**Nhash**	**ABB**
R00	727	726	107,32	121,63	199,23	109,18	749	748	924,9	1886,9	1792,4	904,17	775	774	3604,96	3602,9	3606,12	3603,63
R01	742	741	69,69	98,55	147,33	69,54	759	758	679,22	1532,07	1637,36	589,28	762	761	2178,35	3602,91	3606,01	1793,42
R02	661	660	198,16	196,31	437,05	196,74	688	687	1890,17	3602,88	3606,14	1742,01	691	690	3605,16	3602,89	3606,14	3603,98
R03	691	690	3,49	6,44	6,18	3,46	697	696	4,45	9,8	7,35	4,29	792	791	427,88	486,01	1118,5	426,17
R04	660	659	63,34	71,15	110,93	62,86	685	684	1090,26	1692,56	1934,47	917,24	757	756	3605,14	3602,81	3606,13	3604,71
R05	632	631	175,63	203,38	529,67	172,26	724	723	939,74	3087,72	3027,82	782,26	776	775	3603,26	3602,79	3606,14	3052,79
R06	794	793	113,61	123,27	280,29	111,94	808	807	1174,39	2112,75	2922,84	903,15	825	824	3494,37	3602,74	3606,08	2655,48
R07	594	593	61,72	70,76	143,57	61,15	613	612	514,04	882,08	1397,27	458,14	698	697	348,52	534,53	859,46	341,83
R08	750	749	74,85	107,92	118,2	74,96	766	765	646,35	2080,23	1226,47	613,45	825	825	3605,17	3602,86	3606,15	3604,46
R09	693	692	221	303,31	449,77	219,22	704	703	233,86	437,36	413,56	233,21	740	739	1232,79	3602,85	2545,73	1225,67
R10	664	663	16,8	20,76	29,34	16,69	686	685	286,63	298,45	589,23	282,82	734	733	3604,88	3602,82	3606,14	3603,03
R11	626	625	53,33	45,05	84,9	53,9	682	681	597,33	594,46	1046,54	556,61	726	725	3604,58	3602,79	3606,17	3603,09
R12	742	741	7,32	17,24	10,59	7,45	758	757	8,78	22,48	13,79	8,87	804	803	346,4	641,33	633,76	349,24
R13	695	694	8,18	11,95	13,14	8,29	700	699	14,07	22,76	24,24	14,26	747	746	244,01	372,99	628,05	242,5
R14	681	680	109,73	147	288,27	109,84	727	726	1798,95	3602,74	3606,07	1705,58	766	766	3605,41	3602,8	3606,1	3604,55
R15	629	628	84,43	105,59	167,85	84,67	695	694	101,01	145,89	178,84	99,59	766	765	437,24	822,86	1015,73	438,97
R16	611	610	4,11	4,56	5,71	4,08	643	642	90,64	119,89	179,88	90,32	686	685	285,24	466,42	698,15	286,44
R17	781	780	599,95	566,85	1781,59	554,53	800	799	2411,4	3602,68	3605,97	2037,25	819	818	3605,04	3602,78	3606,06	3604,39
R18	736	735	14,07	21,55	20,08	14,32	749	748	63,12	127,28	97,48	63,47	775	774	829,39	2883,34	1451,15	772,42
R19	790	789	147,84	154,41	383,69	149,18	812	811	833,99	1424,4	1811,91	756,88	837	837	3605	3602,79	3606,14	3604,38
Avg.			**106,73**	**119,88**	**260,37**	**104,21**			**715,17**	**1364,27**	**1455,98**	**638,14**			**1220,68**	**1874,43**	**1797,16**	**1053,18**
Solved			**20**	**20**	**20**	**20**			**20**	**17**	**17**	**20**			**10**	**7**	**8**	**11**

Table 1: Test results of the ABB algorithm and its variants.

find this solution.

We carried out a first set of tests to verify the impact of the polynomial dynamic programming algorithm, the barrier vertex and the hashing strategy, on the performance of the ABB algorithm. To this aim, we tested three variants of the ABB algorithm: *Ndpa*, *Nbar*, and *Nhash* that are the ABB algorithm without the dynamic programming algorithm, the barrier vertex and the hashing strategy, respectively. Table 1 reports the results of the ABB algorithm and of these variants. We tested instances of three different sizes, 25, 27 and 29. For example, the size 25 means 12 pickup and delivery vertices and the depot. The first line shows the size (25, 27, 29) of the tested instances. The first column of the table gives the name of the data set from which the instance is generated. Given a data set and a size s, the instance is generated by taking the first s vertices from each data set.

For each instance, the initial upper bound (UB), the best solution ($Best$) found and the total CPU time, in seconds, spent by each of the algorithms to find the optimal solution, are reported. A time over 3600 indicates that the algorithm was unable to find the optimal solution within one hour. The last two lines show, for each algorithm, the average computational time Avg, taken over the instances solved to optimality by at least one algorithm, and the number of times the optimal solution was found ($Solved$) within the time limit, respectively. We do not compute the average over all instances because the time limit reached when the optimal solution cannot be found within one hour is non-informative of the computational time really needed to reach optimality.

On the instances of size 25, the Ndpa and ABB algorithms require a similar average time, the Nbar is slightly slower, while the Nhash is around 60% slower than the ABB. On the instances with 27 vertices, the gap between ABB and Ndpa increases to 10% and both algorithms solve all the instances to optimality within the time limit. Instead, Nbar and Nhash require more than twice the time required by the ABB and in three cases (R02, R14 and R17) these algorithms fail to find the optimal solution within the time limit. Finally, on the instances with 29 vertices the ABB solves to optimality the largest number of instances, 11, with the smallest average computational time. The variant Ndpa solves 10 instances, Nhash solves 8 instances while Nbar only 7. In terms of computational time, the Ndpa, Nbar and Nhash result, on average, 13%, 43% and 41% slower than the ABB, respectively.

The above comparison shows that the barrier vertex and the hashing strategy are the algorithm charac-

Size →	**25**				**27**				**29**			
Inst. ↓	**#barrier**	**b_best**	**t_best**	**HP**	**#barrier**	**b_best**	**t_best**	**HP**	**#barrier**	**b_best**	**t_best**	**HP**
R00	12	1	0,14	2332669	13	3	263,61	18672256	*4*	*1*	*266,56*	*58393816*
R01	12	1	0,43	1546610	13	1	0,79	11382154	14	1	1,76	34858879
R02	12	2	52,17	4547075	13	2	420,45	41375752	*5*	*2*	*2375,34*	*59305399*
R03	8	3	1,01	49608	9	3	1,18	32535	14	3	84,06	5326858
R04	12	1	0,54	1196169	13	1	42,08	18700250	*3*	*1*	*256,84*	*13856687*
R05	12	1	2,01	3711473	13	1	0,73	14373612	14	2	648	58403667
R06	12	1	9,79	2555311	13	1	125,54	20158123	14	1	253,78	57529531
R07	12	2	6,61	1656589	13	2	42,2	9640423	14	2	45,05	6411029
R08	12	2	14,3	1589626	13	2	116,75	12734862	*2*	*n.d*	*n.d*	*28113971*
R09	12	1	6,81	4418858	13	1	3,88	3658302	10	1	59,97	19975908
R10	12	1	0,56	264023	13	1	38,31	5488285	*14*	*2*	*817,4*	*89384908*
R11	12	2	9,36	608860	13	1	72,81	8138901	*13*	*2*	*496,02*	*66509800*
R12	12	1	0,42	79238	12	1	0,9	106131	14	1	23,9	4829730
R13	12	2	1,96	137001	13	2	3,31	232851	14	2	51,19	4870708
R14	12	1	24,19	2428126	13	1	299,69	43346328	*1*	*n.d*	*n.d*	*20414556*
R15	12	3	40,84	1529953	13	1	19,15	1517196	14	1	61,93	7460913
R16	11	2	1,16	33286	13	2	23,79	1457122	13	6	179,95	4684746
R17	12	2	80,19	11275322	13	2	289,58	45940761	*2*	*2*	*2447,17*	*32080246*
R18	11	1	2,92	192020	12	1	14,59	770993	14	2	255,03	10446746
R19	12	2	39,68	4004924	13	2	129,43	16685116	*2*	*n.d.*	*n.d.*	*29990015*

Table 2: Additional information regarding the behaviour of ABB during the computation.

teristics with the greatest impact on the performance of the ABB while the dynamic programming algorithm for the computation of the delivery tours the one with the smallest impact. We also tested the computational effectiveness of the filters. With a time limit of 1 hour, ABB on 20 instances with 29 vertices finds the optimal solution 11 times while removing the filters it finds the optimal solution 5 times only. Besides, on these 5 instances the time required by the version without the filters is more than five times greater than the time required when the filters are used. This explains why it is important to find as many filters as possible and why we modified the branching strategy, with the introduction of the barrier vertex, to generate new filters.

In order to quantify the impact of the stop criterion, the sequence of barrier vertices chosen and the hashing strategy, we show in Table 2 some relevant information. For each instance, the number of total barrier vertices visited (*#barrier*), the barrier vertex on which the best solution (*b_best*) is found, the time (*t_best*) when this solution is found and the number of pruning operations carried out through the hashing strategy (HP), are reported. The values in italic indicate the instances not solved by ABB within the time limit. The column *#barrier* shows that the stop criterion is successful three times on instances with size 25 and 27 and two times on 29. Therefore, on these instances, on average, in the 13% of cases ABB does not visit all the n barrier vertices. This average grows up to 41% on the smaller instances with size 17, 19, 21 and 23. Column *b_best* reports that in 43% of instances ABB finds the best solution on the first barrier vertex and in 83% within the first three barrier vertices. These results highlight the effectiveness of the chosen sequence. Moreover, from the values of *t_best* column, we find out that, in 81% of cases, the best solution is found within the first 5 minutes of computation. Finally, the column HP reveals the effectiveness of the hashing strategy. On average, ABB performs 2, 13 and 30 millions of pruning operations on the instances of size 25, 27 and 29, respectively.

From the results of Table 1 we know what are the instances hardest to solve for the ABB. Since the DTSP2S can be solved in reverse order, that is starting the construction of the tour from the delivery vertices, it is interesting to investigate whether the hardness of instances changes if solved in reverse order. We implemented a variant of ABB, denoted as ABB_rev, that builds the tour in reverse order. In Table 3

Size →	25		27		29	
Inst. ↓	**ABB**	**ABB_rev**	**ABB**	**ABB_rev**	**ABB**	**ABB_rev**
R00	109,18	24,14	904,17	283,59	3603,63	1663,65
R01	69,54	36	589,28	407,06	1793,42	1518,5
R02	196,74	104,97	1742,01	615,45	3603,98	951,69
R03	3,46	11,36	4,29	84,2	426,17	3613,21
R04	62,86	222,46	917,24	2453,76	3604,71	3614,68
R05	172,26	506,7	782,26	3613,41	3052,79	3614,68
R06	111,94	230,49	903,15	2401,16	2655,48	3614,56
R07	61,15	70,3	458,14	2091,42	341,83	2808,89
R08	74,96	15,93	613,45	86,4	3604,46	788,28
R09	219,22	2,24	233,21	3,25	1225,67	39,01
R10	16,69	79,62	282,82	982,07	3603,03	3614,45
R11	53,9	50,37	556,61	1366,6	3603,09	3614,56
R12	7,45	17,68	8,87	11,01	349,24	1416,33
R13	8,29	11,55	14,26	93,82	242,5	3125,75
R14	109,84	50,58	1705,58	1143,09	3604,55	3614,7
R15	84,67	162,74	99,59	231,54	438,97	3614,11
R16	4,08	87,6	90,32	2410,67	286,44	3614,58
R17	554,53	1211,29	2037,25	3613,93	3604,39	3614,72
R18	14,32	11,74	63,47	219,38	772,42	1159,05
R19	149,18	11,71	756,88	49,88	3604,38	836,5
Avg.	**104,21**	**145,97**	**638,14**	**1108,08**	**2201,06**	**2522,6**

Table 3: Performance comparison between the ABB and the reverse version that build the tour starting from delivery vertices.

we compare the performance of this variant with the original algorithm.

The results show that often the computational time changes according to the direction of search. It is interesting to highlight that ABB_rev is able to solve within the time limit four instances (R00, R02, R08 and R19) on which the original one fails. Unfortunately, given a generic instance, we cannot establish, a priori, which search direction is the most effective one.

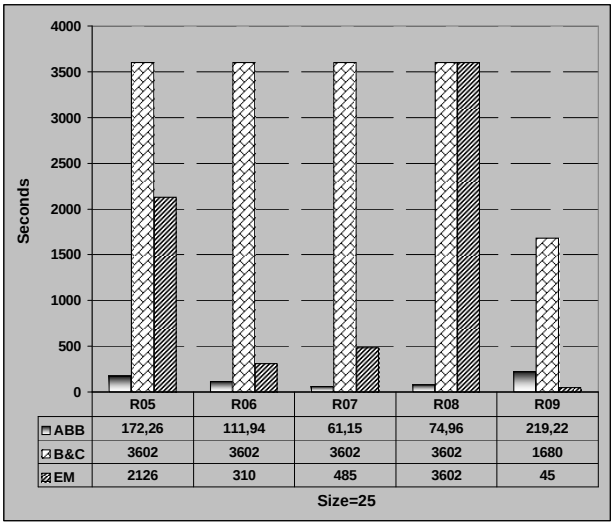
From the results, it is evident that the heuristic upper bound used is often optimal. In order to test the sensitivity of the algorithm to the quality of this UB, we have increased the value of the UB by 5% and ran the algorithm on the largest instances (29 vertices). The results show that the instances solvable to optimality within the time limit remain the same, with the exception of R05, that is not solved when the UB is increased. The average CPU time, for the solved instances, increases of 20%. Having completed the description of ABB characteristics, we compare now the performance of this algorithm with the others available in the literature. To the best of our knowledge, there are only two exact approaches for the DTSP2S. The approach proposed by Lusby et al. [20], that we indicate by EM, is based on the generation of pairs of pickup and delivery tours of minimum cost and on verifying whether a feasible loading configuration exists for these tours. A branch-and-cut approach to different mathematical programming formulations of the problem was proposed by Petersen et al. [21]. We indicate the branch-and-cut applied to the most effective formulation by B&C.

In Table 4 we compare the performance of the ABB with the performance of these two algorithms. The

Inst.	Size	ABB		B&C		EM	
		Best	**Time**	**Gap**	**Time**	**Gap**	**Time**
R05	17	501	**0,25**	0%	1	n.d.	n.d.
R06	17	694	**0,51**	0%	31	n.d.	n.d.
R07	17	487	**0,56**	0%	27	n.d.	n.d.
R08	17	642	**0,57**	0%	38	n.d.	n.d.
R09	17	558	**0,76**	0%	17	n.d.	n.d.
R05	21	546	**2,28**	0%	196	0%	4
R06	21	774	5,66	0%	678	0%	**5**
R07	21	547	1,60	0%	115	0%	**1**
R08	21	670	**2,00**	0%	392	0%	5
R09	21	610	4,65	0%	44	0%	**1**
R05	25	631	**172,26**	7,92%	3602	0%	2126
R06	25	793	**111,94**	2,77%	3602	0%	310
R07	25	593	**61,15**	4,05%	3602	0%	485
R08	25	749	**74,96**	4,65%	3602	0,13%	3602
R09	25	692	219,22	0%	1680	0%	**45**
R05	29	775	**3052,79**	7,08%	3602	1,98%	3602
R06	29	824	**2655,48**	5,22%	3602	0,73%	3602
R07	29	697	**341,83**	6,95%	3602	2,20%	3602
R08	29	825	3604,46	8,01%	3602	5,73%	3602
R09	29	739	1225,67	1,53%	3602	0%	**211**
Avg.			**566,54**		**2280,07**		**1257,21**
Solved			**19**		**11**		**15**

Table 4: Performance comparison between the EM, B&C and ABB algorithms.

first two columns report the name of the data set and the size of the instance, respectively. Since the B&C was tested only on 5 instances (R05-R09) out of 20 available and using the sizes 17, 21, 25 and 29, in Table 4 only the results for these instances and sizes are reported. The column ABB is divided in two sub-columns: Best and Time. The former shows the solution value computed by the algorithm and the latter the total CPU time, in seconds, spent to compute this solution. The columns B&C and EM are divided in two sub-columns: Gap and Time. The former shows the gap, in percentage, between the solution value found by the algorithm and the optimal value while the latter is the total CPU time, in seconds, spent to compute this solution. In each line we indicate in bold the fastest algorithm. Finally, the last line shows the number of times the optimal solution was found (*Solved*) within the time limit. The smallest instance used for the tests has 17 vertices. Unfortunately, for this size no result for the EM algorithm is available. The comparison between the ABB and the B&C reveals a significative difference in terms of computational time. The ABB algorithm solves all the instances in less then one second while the B&C requires at least 17 seconds, with the exception of R05. This behavior is emphasized when the size of the instances increases. On the instances with 21 vertices, the time required by the B&C ranges from 44 to 678 seconds while ABB never requires more than 6 seconds. Instead, the performance of the EM algorithm is similar to that of the ABB. When we consider the instances with 25 vertices the B&C finds the optimal solution only once (R09) within the time limit. On this instance the time required is 1680 seconds, approximately eight times the 214 seconds required by the ABB. On the remaining instances, R05-R08, the quality of the solutions found by the B&C



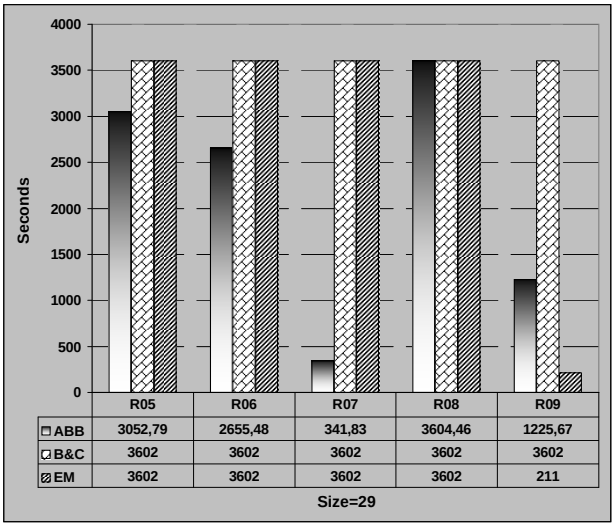


Figure 9: Histograms of three exact approach performance available in literature on the size 25 and 29.

is quite poor with a percentage gap that ranges from 2,77% to 7,92%. The EM fails once, on R08, while on R09 it is faster than the ABB. On the other three instances it is from 3 to 12 times slower than the ABB. Finally, on the largest instances with 29 vertices, the B&C never finds the optimal solution while the EM does only once on R09 where it is faster than the ABB. On average, the gap from the optimal solution for the B&C and EM is around 5,75% and 2,66%, respectively. In the worst case, this gap reaches 8% for the B&C and almost 6% for the EM. With the exception of R08, the ABB solves all the instances in one hour. Since no results are available for the EM on size 17, we computed the average from size 21. The Avg line gives an idea of the different performance of the three algorithms, although we have to take into account that the several 3602 values that contribute to the average (in particular for B&C) tend to flatten the average. The last line of the table shows that on 20 instances the ABB, B&C and EM solve 19, 11 and 15 of them in one hour, respectively. The histograms depicted in Figure 9 give a graphical representation of the three algorithms performance on the size 25 and 29 and show that ABB outperforms the other two approaches in terms of computational time.

The tests presented by Lusby et al. [20] are carried out on all the 20 kinds of instances R00-R19 with a time limit equal to 3 hours. In order to have a complete comparison on all the instances available, we fixed the same time limit to 3 hours for the ABB and we ran again all the tests.

Table 5 shows the new results. On the instances with 21 vertices, with the exception of some particular case like R10 and R16, the performance of the two algorithms is similar. The different average is essentially due to the bad result on R16 produced by EM. On 25 vertices the difference starts to be relevant. Here, there is a first failure of the EM algorithm to find the optimal solution on instance R08. Moreover, only in three cases (R01, R09, R12) the EM is faster than the ABB. On the remaining 17 instances, the ABB turns out to be at least 10 times faster than the EM in 6 cases (R02, R04, R05, R08, R10 and R16) with an impressive time gap on R08, R10 and R16. The Avg line gives a clear idea of the different level of efficiency of the two algorithms. On average, the ABB is around 14 times faster than the EM. Finally, on the size 29 the different ability of the two algorithms to find the optimal solution within the time limit becomes evident. The EM only in five cases finds this solution while the ABB 16 times. Moreover, only in two cases (R01,

Size →	**21**		**25**		**29**	
Inst. ↓	**EM**	**ABB**	**EM**	**ABB**	**EM**	**ABB**
R00	**5**	5,12	142	**108,48**	10802	**7592,21**
R01	**3**	4,11	**17**	68,51	**653**	1787,94
R02	**6**	7,57	2432	**197,68**	10802	**5638,48**
R03	1	**0,98**	4	**3,43**	10802	**423,42**
R04	3	**1,28**	1151	**62,28**	10802	**6437,22**
R05	4	**2,29**	2126	**173,91**	10802	**3040,04**
R06	**5**	5,72	310	**109,72**	10802	**2657,65**
R07	**1**	1,61	485	**61,8**	10802	**339,85**
R08	5	**2,02**	10802	**75,36**	10802	10802
R09	**1**	4,68	**45**	220,29	**211**	1209,25
R10	7	**0,88**	2452	**16,62**	10802	**3924,84**
R11	2	**0,56**	356	**53,16**	10802	**3658,66**
R12	3	**1,72**	**7**	7,32	10802	**343,56**
R13	2	**1,3**	16	**8,1**	1499	**240,62**
R14	13	**4,78**	205	**110,49**	10802	10802
R15	**2**	2,52	306	**85,37**	1152	**436,3**
R16	114	**0,99**	3537	**4,04**	10802	**284,46**
R17	11	**4,52**	3832	**551,97**	10802	10802
R18	1	**0,58**	16	**14**	3607	**757,32**
R19	1	**0,58**	171	**150,8**	10802	10802
Avg.	**9,50**	**2,69**	**1420,60**	**104,17**	**7871,50**	**2423,24**
Solved	**20**	**20**	**19**	**20**	**5**	**16**

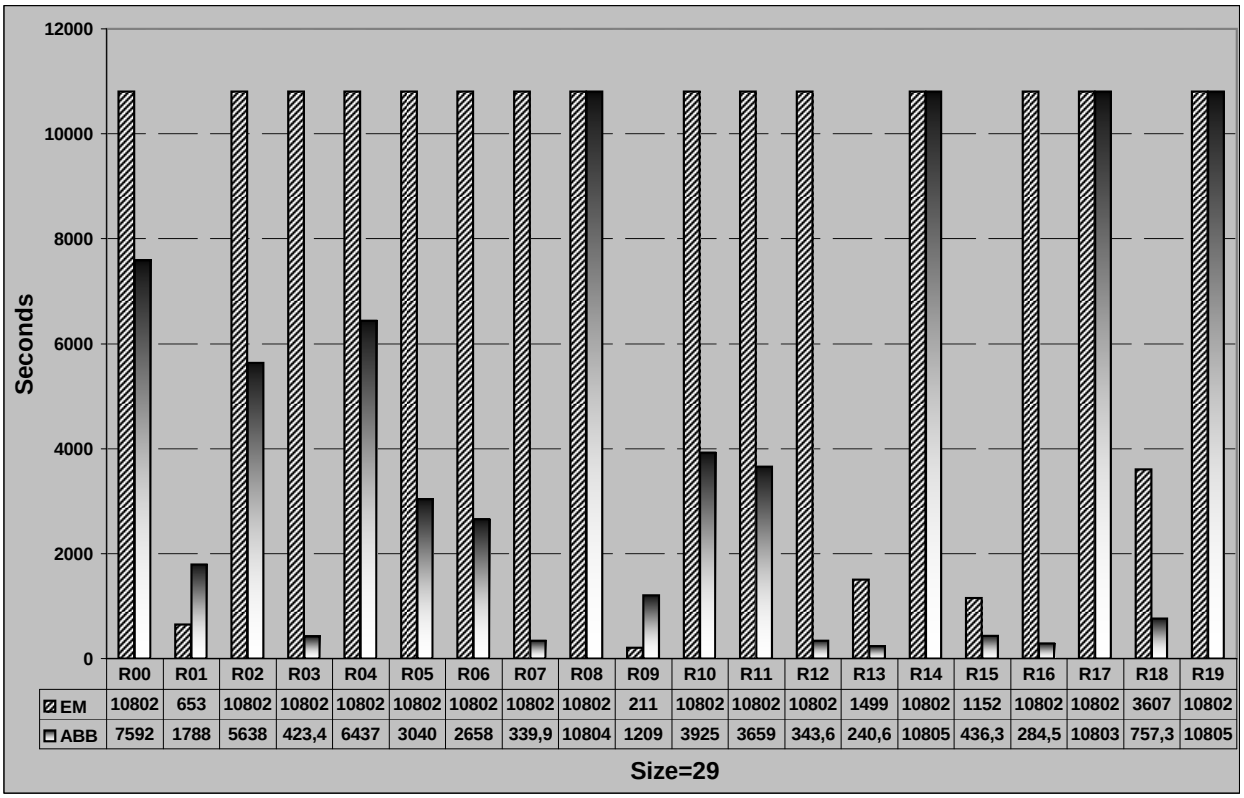
Table 5: Comparison between the EM and ABB algorithms on all 20 instances with a time limit equal to 3 hours.

R09) the EM is faster than the ABB. Notice that from Avg line the ABB turns out to be around 4 times faster than the EM. The histogram of the performance of these two algorithms is depicted in Figure 10.

Finally, using the threshold of three hours and keeping track of the time when ABB finds the best solution, we discovered that only on R08, R14 and R19 the algorithm does not obtain the optimal solution within one hour, while we were unable to solve R17 within the three allowed hours and thus cannot state anything. This means that 16 out of 20 solutions obtained by the ABB, on 29 vertices, are optimal. This analysis suggests that there is a high probability for the ABB to obtain the optimal solution within the time limit, though possibly without optimality proof.

5. Conclusions

In this paper we have presented an additive branch-and-bound algorithm for the solution of the Double TSP with Two Stacks. The problem is one of the most interesting combined routing and loading problems. Although the problem is simple to describe, it is computationally extremely hard. Previous papers have presented different formulations for the problem, solved with a branch-and-cut approach, and an exact approach based on the enumeration of pairs of pickup and delivery tours. The algorithm we present in this paper outperforms, on a set of benchmark instances, the previous exact approaches in terms of computational



	R00	R01	R02	R03	R04	R05	R06	R07	R08	R09	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19
EM	10802	653	10802	10802	10802	10802	10802	10802	10802	211	10802	10802	10802	1499	10802	1152	10802	10802	3607	10802
ABB	7592	1788	5638	423,4	6437	3040	2658	339,9	10804	1209	3925	3659	343,6	240,6	10805	436,3	284,5	10803	757,3	10805

Figure 10: Performance comparison between EM and ABB using a time limit equal to 3 hours.

time and number of optimal solutions. This is due to the ideas, like the barrier vertex and the hashing strategy, that, embedded into the additive approach, allowed us to raise to 29 the size of the instances solvable to optimality within one hour. Moreover, test results reveal that, even when the ABB reaches the time limit, often the solution returned is the optimal one. The size of the instances solved to optimality remains quite small, confirming the fact that even the simplest combined routing and loading problems can be computationally extremely hard.

In this work we focused the attention on the case with two stacks. Although the ABB can work for any number of stacks, when the number of stacks increases, the effectiveness of the filters decreases and the performance of the algorithm worsens. Indeed, on the instances R05-R09 with size 12 and three stacks, ABB turns out to be faster than B&C in two cases. However, increasing the size to 15, ABB never finds the optimal solution within the time limit while the B&C always does. Comparison with results reported in [20] and [21] reveals that, currently, the EM algorithm is the fastest exact approach for DTSPMS with three and four stacks. A possible direction for future work is to improve the quality of the lower bounds or to introduce new ideas that improve the performance of the algorithm when the number of stacks is greater than two.

Acknowledgments

We acknowledge the comments and suggestions made by two anonymous referees that helped us to significantly improve an earlier version of this paper.

References

[1] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows*. Prentice Hall, 1993.

[2] M. Battarra, G. Erdogan, G. Laporte, and D. Vigo. The travelling salesman problem with pickups, deliveries and handling costs. *Transportation Science*, 2010. To appear.

[3] 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, 2007.

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

[5] M. Casazza, A. Ceselli, and M. Nunkesser. Efficient algorithms for the double traveling salesman problem with multiple stacks. In *Proceedings of Cologne Twente Workshop (CTW 2009)*, pages 7–10, 2009.

[6] J.-F. Cordeau, M. Dell'Amico, and M. Iori. Branch-and-cut for the pickup and delivery traveling salesman problem with fifo loading. *Computers & Operations Research*, 37:970–980, 2010.

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

[8] 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, 2009.

[9] K. Doerner, G. Fuellerer, M. Gronalt, R. Hartl, and M. Iori. Metaheuristics for the vehicle routing problem with loading constraints. *Networks*, 49:294–307, 2007.

[10] J. Edmonds. Optimum branching. *Journal Research of the National Bureau of Standards*, 71B:233–240, 1967.

[11] G. Erdogan, J.-F. Cordeau, and G. Laporte. The pickup and delivery traveling salesman problem with first-in-first-out loading. *Computers & Operations Research*, 36:1800–1808, 2009.

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

[13] M. Fischetti and P. Toth. An additive bounding procedure for combinatorial optimization problems. *Operations Research*, 37:319–328, 1989.

[14] M. Fischetti and P. Toth. An additive branch and bound procedure for the asymmetric tsp. *Mathematical Programming*, 53:173–197, 1992.

[15] G. Fuellerer, K.F. Doerner, R.F. Hartl, and M. Iori. Ant colony optimization for the two-dimensional loading vehicle routing problem. *Computers & Operations Research*, 36:655–673, 2009.

[16] G. Fuellerer, K.F. Doerner, R.F. Hartl, and M. Iori. Metaheuristics for vehicle routing problems with three-dimensional loading constraints. *European Journal of Operational Research*, 201:751–759, 2010.

[17] M. Gendreau, M. Iori, G. Laporte, and S. Martello. A tabu search algorithm for a routing and container loading problem. *Transportation Science*, 40:342–350, 2006.

[18] M. Gendreau, M. Iori, G. Laporte, and S. Martello. A tabu search heuristic for the vehicle routing problem with two-dimensional loading constraints. *Networks*, 51:4–18, 2008.

[19] M. Iori, J.J. Salazar Gonzalez, and D. Vigo. An exact approach for the vehicle routing problem with two-dimensional loading constraints. *Transportation Science*, 41:253–264, 2007.

[20] R. Lusby, J. Larsen, M. Ehrgott, and D. Ryan. An exact method for the double tsp with multiple stacks. *International Transactions in Operational Research*, 17(5):637–652, 2010.

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

[22] H.L. Petersen and O.G. Madsen. The double travelling salesman problem with multiple stacks - formulation and heuristic solution approaches. *European Journal of Operational Research*, 198:139–147, 2009.

[23] R.E. Tarjan. Finding optimum branchings. *Networks*, 7:25–35, 1977.

[24] S. Toulouse and R. Wolfler-Calvo. On the complexity of the multiple stack TSP, kSTSP. In *Lecture Notes in Computer Science, Proceedings of the 6th Annual Conference on Theory and Applications of Models of Computation*, volume 5532, pages 360–369, 2009.

[25] F. Tricoire, K. Doerner, and R. Hartland M. Iori. Heuristic and exact algorithms for the multi-pile vehicle routing problem. *OR Spectrum*, 2010. To appear.