

The Very Offline k -Vehicle Routing Problem in Trees

Ion Muslea

Information Sciences Institute/University of Southern California
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292-6695, United States of America
muslea@isi.edu

Abstract

The vehicle routing problem (VRP) in trees is a restriction of the general vehicle routing problem in which the underlying graph is a tree. In this paper, we introduce several NP-complete variants of the problem (e.g., common-source, common-destinations, in-place, flexible-requests, and bring-it-back), and we present five fast approximation algorithms for these variants. We mainly focus on the 2-vehicle routing problems, but we also consider possible generalizations for the k -vehicle routing case. In our analysis, the underlying graphs consist of various types of trees: binary and non-binary, weighted and unweighted.

1. Introduction

Routing problems involve the movement of a given set of objects from their original positions to their respective destinations. Such problems have a wide range of applications, from robotics to AI planning to mail delivery. While trying to solve a routing problem, one has to keep in mind various factors like cost optimization, service improvement, and time or ordering constraints. Graphs represent useful tools in the analysis of the routing problems because they provide a level of abstraction that facilitates the modelling process.

Let $G = (V, E)$ be an undirected, weighted graph with the set of vertices V and the set of edges E , and let O be a set of objects distributed in the vertices of G . Given k mobile servers located in the nodes of G (each server being able to transport only one object at the time) and a set R of requests $(source_i, destination_i)$ (each request being associated to a unique object $o_i \in O$), the k -vehicle routing problem (k VRP) is concerned with the movement of the servers such that each object o_i is transported by a server from its initial position $source_i$ to its wanted position $destination_i$. After serving the requests, the servers must return to their original location. The goal of the problem is

to minimize the total distance traveled by the servers during the process of transporting the objects.

Frederickson, Hecht, and Kim [9] have proved the vehicle routing problem to be NP-complete for the case of general graphs. As many of the NP-complete graph-problems have interesting properties when the underlying graph is a tree (see [2], [7], [8], and [11]), we shall study the vehicle routing problem in different types of trees (e.g., binary and non-binary, weighted and unweighted).

A possible way to differentiate among the several variants of the vehicle routing problem is based on the initial knowledge about the problem. With regard to this criteria, we have three categories of algorithms: online algorithms, offline algorithms, and very offline algorithms. The *online algorithms* get the requests one at the time and must serve the incoming request without waiting for the following requests. Conversely, *offline algorithms* know from the very beginning the entire set of requests, together with a pre-established order in which the requests have to be served. Finally, the *very offline algorithms* (see [14]) are a special case of offline algorithms in which the requests may be served in any particular order.

In this paper we will be concerned with the very offline version of the vehicle routing problem (from now on, by both VRP and k VRP we will denote the very offline variant of VRP). Even in this case, there are several possible ways to define and interpret a request, and, consequently, there are different variants of the very offline vehicle routing problem. Given a request $(source_i, destination_i)$, a server has to pick an object from $source_i$ and to deliver it to the $destination_i$. But such a request may be treated in two distinct ways: *preemptive* (the server may drop the object in an intermediate node and come later to pick it up) and *non-preemptive* (once an object is picked from its source it has to be delivered directly to the destination).

A degenerate case of the k VRP arises when for each request $(source_i, destination_i)$ the source and the destination represent the same vertex of the graph (that is, $source_i = destination_i$). This problem is called the *in-*

place vehicle routing problem, and it corresponds to the real-world case when the server has to visit some locations and to do some in-place maintenance/processing work.

Two interesting variants of the vehicle routing problem are concerned with the cases when all the sources or all the destinations are located in the root of the tree. They are called the *common-source*, respectively the *common-destination* VRP. Another variant of the vehicle routing problem deals with *flexible requests*. A flexible request is a request defined as $(\{src_1, src_2, \dots, src_m\}, \{dest_1, dest_2, \dots, dest_n\})$, and in order to serve such a request, the server has to pick an object from any of the m sources and to deliver it to any of the n destinations. By restricting to one the number of sources, respectively destinations, in each request, the flexible requests problem becomes the *multiple-destinations*, respectively *multiple-sources*, problem.

Finally, in the *bring-it-back* vehicle routing problem the server that serves a requests $(src_i, dest_i, T_i, D_i, P_i)$ must pick the object o_i from src_i , deliver it to $dest_i$ (where the object is processed in time T_i), and bring it back to src_i . If the object o_i does not arrive back in src_i before the deadline D_i , the server has to pay a penalty P_i .

2. Literature review

Unfortunately, many routing problems are NP-complete in the sense of Cook [4] and Karp [15]. Frederickson, Hecht, and Kim [9] proved that the k -traveling-salesman problem, the k -chinese-postman problem, and the k -stacker-crane problem are NP-complete if the underlying graph has no special properties. As it has been conjectured that there are no polynomial-time, optimal solutions for NP-complete problems (see [5] and [10]), researchers focused on two main directions: finding polynomial-time approximation algorithms with good performance ratios [10], and identifying graphs with special properties that make the problems easier to solve.

Frederickson, Hecht, and Kim [9] presented a mixed-strategy approximation algorithm that can be successfully used to solve several versions of the vehicle routing problem in a general graph. By combining two stand-alone algorithms, each of which being efficient in some of the extreme cases, the final solution improves the performances of its individual components. These algorithms are based on a tour-splitting heuristic and run in cubical time.

By analyzing the preemptive version of the 1-vehicle routing problem in trees, Frederickson and Guan [8] showed that the problem can be optimally solved in sub-quadratic time. As the non-preemptive version of the 1-vehicle routing problem in trees is NP-complete [7], the authors introduced a linear-time approximation algorithm with a performance ratio of $3/2$, and a slightly slower approximation algorithm

with a performance ratio of $5/4$. Furthermore, the paper also presented a family of approximation algorithms with a performance ratio going down to 1.21363 as the running time increases. Frederickson [6] also introduced approximation algorithms for the 1-vehicle routing problem in circular and linear graphs, and Attalah and Kosajaru [1] provided efficient solutions to a transportation problem based on a robot arm that can make circular and “telescoping” movements. Finally, Chobrak and Larmore [3] designed several fast algorithms for the two-server problem.

A special case of the in-place 1-vehicle routing problem in trees deals with objects that have associated a release time r_i (i.e., the objects cannot be processed before the moment denoted by r_i) and a handling time h_i (i.e., the server has to spend h_i time to process the corresponding object). Karuno, Namagochi, and Ibaraki [11] proved this problem to be NP-complete and showed that an additional depth-first constraint makes the problem solvable in $O(|V| \times \lg(|V|))$. Furthermore, by using the same algorithm as an approximation algorithm for the case when the depth-first constraint does not exist, the authors proved that its worst-case ratio is at most two. A detailed analysis of the routing and scheduling problems that have time window constraints is done in by Solomon [17], and Solomon and Desrosiers [18].

The k -sales-delivery-man problem (see [2]) is a combination of the k -traveling-salesman problem and the k -delivery-man problem, and it deals with the minimization of both the total working time of the server and the waiting time of the objects, the first criteria being more important than the second one. The authors analyzed the problem for the cases when the underlying graphs are trees and cactus-networks, and they presented polynomial time solutions for the 1-vehicle routing variant.

Klostermeyer [14] proved that the in-place, very offline k -server problem in graphs is NP-complete and presented a Steiner-tree-based approximation algorithm for the 1-vehicle routing problem. This algorithm has a performance ratio of $2B$, where B is the best known performance ratio for Steiner trees. The same paper also introduced a family of approximation algorithms (performance ratio = $2B + \epsilon$) for the 2-server problem. Klostermeyer (see [13] and [12]) also studied the online k -server problem in dynamically orientable graphs.

3. Definitions

An undirected graph $G = (V, E)$ consists of a finite set V of vertices and a set E of pairs of vertices, where each element of E contains two distinct vertices. The set V is called the vertex set of G , while the set E is called the edge set of G . Each edge $(u, v) \in E$ connects the vertices $u, v \in V$, and the edge (u, v) is the same as the edge (v, u) .

A vertex v is incident with the edge (u, w) if and only if either $v = u$ or $v = w$. The *degree* of a vertex v is equal to the number of edges incident with v .

A weighted graph is a graph G for which each edge has associated a weight given by a weight function $w : E \rightarrow \mathbb{R}$. We will denote the weight of the edge (u, v) by $w(u, v)$. A subgraph $H = (U, F)$ of $G = (V, E)$ is a graph such that $U \subseteq V$, $F \subseteq E$, and $\forall (u, v) \in F$ we have $u, v \in U$.

A *path* P in the graph $G=(V,E)$ is a sequence of vertices v_1, v_2, \dots, v_n such that $\forall i \in [1, n - 1], (v_i, v_{i+1}) \in E$. The vertex v_1 is called the *initial vertex*, and the vertex v_n is called the *terminal vertex*. A *cycle* C is a path in which the initial and the terminal vertices are the same, while a *tour* T is cycle that visits all the vertices in V . A *subtour* ST is a tour visiting a given subset of vertices $W \subset V$ and a *k-tour* consists of a set of k subtours having the same starting point and collectively visiting all vertices in V .

An undirected graph $G = (V, E)$ is *connected* if and only if there is a path P between every pair of vertices from V . A *tree* is a connected, acyclic, undirected graph with a distinct vertex r called the root of the tree. In trees, the vertices of the graph are also called nodes. Any node v in the path from the root to a given vertex u is called an *ancestor* of u , while u is a *descendant* of v . If v is an ancestor of u and $(u, v) \in E$, we say that v is the *parent* of u , and u is the *child* of v . A node with no descendants is a *leaf*, while a node having at least one child is called an *internal node*. The length of the path from the root to a node v is called the *depth* of v , and the *height* of the tree is the largest depth of a node in the tree. A *subtree* of the tree T is a subgraph rooted at a vertex v of T and consisting of all descendants of v together with all edges on the paths from v to its descendants. Finally, a *binary tree* is a tree in which each node has at most two children.

4. Complexity Results

The ‘‘classic’’ 1-vehicle routing problem in trees is defined as follows. Let $T = (V, E)$ be a tree, and let $R = \{r_1, r_2, \dots, r_n\}$ be a set of requests, where $r_i = (source_i, destination_i)$, and both $source_i$ and $destination_i$ are nodes of the tree T . Given a unique server that is initially located in the root and can carry one object at the time, we want to find the order in which the requests must be served such that the total distance travelled by the server is minimized.

Frederickson and Guan [8] showed that the preemptive problem can be optimally solved in polynomial time. In the companion paper [7], the same authors proved that the non-preemptive version of the problem is NP-complete, and they presented several approximation algorithms (one of them running in linear time) with excellent performance ratios.

Problem	Type	Running Time
in-place		$O(V + R)$
common-src	n-p, $src = root$	$O(V \times R)$
common-src	n-p, $src \neq root$	$O(V + R ^2)$
common-src	p, $src = root$	$O(V + R)$
common-dest	n-p, $src = root$	$O(V \times R)$
common-dest	n-p, $src \neq root$	$O(V + R ^2)$
common-dest	p, $src = root$	$O(V + R)$

Table 1. Running times of 1-VRP algorithms

In [16], we analyzed several versions of the 1-vehicle routing problem in trees, and we introduced a variety of fast polynomial-time algorithms that solve those problems. As we can see in Table 1, the running times of all our solutions are either linear (e.g., in-place 1-VRP) or - at most - quadratic (several variants of the common-source and common destination 1-VRP).

In order to show that several variants of the k -vehicle routing problem are NP-complete, first we have to define the k -vehicle routing problem as a decision problem:

INSTANCE: Let us have a tree $T = (V, E)$, a finite set R of requests (each request being defined as a pair $(source, destination)$, where both $source$ and $destination$ are nodes in T), a number k of servers located in the root of the tree, and a ‘‘threshold distance’’ L .

QUESTION: Is there a partition $\{R_1, R_2, \dots, R_k\}$ of R and an order in which the servers s_i can serve all the requests from R_i and return to the root without traveling a distance greater than L ?

Many of our NP-completeness proofs are based on transformations from the Multiprocessor Scheduling Problem (MSP) or the Two-Processor Scheduling Problem (2PSP), which are defined in [10]. We present now our NP-completeness results from [16], and for each theorem and corollary we specify the main idea behind its proof.

Corollary 1 *The non-preemptive kVRP in trees is NP-complete.*

Proof: restriction to non-preemptive 1-vehicle routing problem. This result is an obvious corollary to the theorem presented by Frederickson and Guan in [7].

Theorem 1 *The preemptive kVRP in trees is NP-complete for any $k \geq 2$.*

Proof: transformation from MSP.

Theorem 2 *The in-place, kVRP in trees is NP-complete.*

Proof: transformation from MSP.

Theorem 3 *The in-place 2-VRP in binary, unweighted trees is NP-complete.*

Proof: transformation from 2PSP.

Note: the proof of this theorem was by all means the most challenging one among all NP-completeness results presented in this paper. Due to space restrictions, we can not present here our proof, but the interested reader can find it in [16].

Corollary 2 *The in-place k VRP in binary, unweighted trees is NP-complete.*

Proof: restriction to in-place 2-VRP.

Corollary 3 *The k VRP in binary, unweighted trees is NP-complete.*

Proof: restriction to the in-place k VRP in binary, unweighted trees.

Theorem 4 *The common-source k VRP in trees is NP-complete.*

Proof: transformation from MSP.

Theorem 5 *The common-source k VRP in binary, unweighted trees is NP-complete.*

Proof: transformation from MSP.

Theorem 6 *The common-destination k VRP in trees is NP-complete.*

Proof: transformation from MSP (similar with the proof of Theorem 4).

Theorem 7 *The common-destination k VRP in binary, unweighted trees is NP-complete.*

Proof: transformation from MSP (similar with the proof of Theorem 5).

Theorem 8 *The non-preemptive, flexible-requests k VRP in trees is NP-complete.*

Proof: restriction to the non-preemptive k VRP.

Theorem 9 *The preemptive, flexible-requests k VRP in trees is NP-complete for any $k \geq 2$.*

Proof: restriction to the preemptive k VRP.

Corollary 4 *Both the preemptive and the non-preemptive, flexible-requests k VRP in binary, unweighted trees are NP-complete for $k \geq 2$.*

Proof: restriction to the in-place k VRP.

Corollary 5 *The bring-it-back k VRP in trees is NP-complete for any $k \geq 2$.*

Proof: restriction to the in-place k VRP.

Corollary 6 *The bring-it-back k VRP in binary, unweighted trees is NP-complete for any $k \geq 2$.*

Proof: restriction to bring-it-back k VRP.

5. General-Purpose Algorithms

As it is believed that NP-complete problems can not be optimally solved in polynomial time, in the following sections we present a collection of approximation algorithms for 2-VRP.

In order to introduce our approximation algorithms, we begin by presenting four general-purpose routines that are heavily used within the next sections: *PruneTree()*, *FindPath()*, *ComputeDistances()*, and *TraverseSubtree()*.

A tree may contain branches that do not have to be traversed in order to serve the requests. For instance, if a leaf is not a “useful node” (i.e., the source or the destination of at least one request), the edge that leads to it can be removed without affecting the way in which we serve the requests. In this paper, by “pruning a tree” we denote the operation of removing all subtrees that do not include “useful nodes”.

Our pruning algorithm starts by identifying the list of all leaves in the tree and continues by analyzing every single leaf in the list. If the leaf is not a useful node, it is removed from the tree. After removing a leaf, we also analyze its parent to see whether the node removal turned the parent into a leaf (in such a case we add the parent to the list of leaves).

PruneTree(tree $T = (V, E)$, requests-list R)

- create a list L of all leaves in T
- determine the parent of each node in T
- traverse R and mark all the useful nodes in T
- FOR EACH leaf $l \in L$ DO
 - IF l is not a useful node THEN
 - let v be the parent of l
 - $V = V - \{l\}$
 - $E = E - \{(v, l)\}$
 - IF $\text{degree}(v) == 1$ THEN $L = L \cup \{v\}$
- return T

By analyzing the algorithm above, we can see that the pruning operation is done in $O(|V| + |R|)$. As the pruning is a fast operation that simplifies the problem at hand, we will perform it on all the input-trees of our approximation algorithms.

The other three general-purpose routines are quite trivial, and we will just explain their behavior without providing the

pseudo-code. For a given tree, *FindPath()* finds (in $O(|V|)$) the paths from the root to each node, while *ComputeDistances()* is used to compute the distances between a given node and all other nodes in the tree. Finally, *TraverseSubtree()* runs in $O(|E|)$ and computes the distance traveled by a server that traverses a given subtree.

6. Approximation Algorithms for the in-place 2-VRP

In [16], we have shown that a server that optimally solves the in-place problem in a tree has to travel a distance equal to twice the weight of the pruned tree. In a best case scenario, an optimal solution for the in-place 2-VRP would equally split the distance traveled by a unique server between the two available servers. It follows that such an optimal solution would make each server travel at least a distance equal to the weight of the pruned tree.

As we shall see, all the approximation algorithms presented in this section generate solutions such that each server travels at most a distance equivalent to twice the weight of the pruned tree. Consequently, all the three algorithms below have an absolute performance ratio $R_A \leq 2$. We were not able to prove a better result, but our experimental data (see the results presented in Experimental Results section) show that in most of the cases our approximation algorithms have performance ratios close to 1.0.

6.1. The *RemoveEdge()* Algorithm.

In order to partition the set of requests in two subsets, the *RemoveEdge()* algorithm splits the pruned tree T_0 in two subgraphs T_1 and T_2 , and makes the servers s_1 and s_2 serve the requests located in T_1 , respectively T_2 . The tree splitting is done by removing an edge $(v, parent(v))$ and by assigning the requests in the v -rooted subtree to s_1 , and all the other ones to s_2 . Each server will serve the requests that were assigned to it by traversing the corresponding tree in a depth-first manner. Our algorithm considers all possible partitions by successively removing each edge, computing the traveled distance, and then adding back the removed edge.

During the depth-first traversal of T_i , the server s_i travels a distance equivalent to twice the weight of T_i . As the edges of T_i represent a subset of the edges of T_0 , it follows that each server travels a distance shorter than $2 \times weight(T_0)$, and, consequently, the absolute performance ratio of *RemoveEdge()* is $R_A \leq 2$. By analyzing the pseudo-code below, it is easy to see that the total running time for the *RemoveEdge()* approximation algorithm is $O(|V| + |E|^2)$.

RemoveEdge(tree $T = (V, E)$, requests-list R)

- let r be the root of T

- let D_1, D_2 be the distances traveled by s_1, s_2
- $T_0 = PruneTree(T, R)$
- $FindPaths(r, T_0)$
- $ComputeDistances(r, T_0)$
- FOR EACH edge $(v, parent(v))$ DO
 - $D_1 = 2 \times dist(r, v) + TraverseSubtree(v, T_0)$
 - remove the edge $(parent(v), v)$ from T_0
 - $D_2 = TraverseSubtree(r, T_0)$
 - add back the edge $(parent(v), v)$ to T_0
 - IF $\max(D_1, D_2)$ is a new minimum THEN
 - update the minimum information
- return

By simultaneously removing $k - 1$ edges from the tree, we can modify the algorithm above such that it works for k servers. As we have to consider all combinations of $k - 1$ edges from a total of $|E|$ edges, the running time will be rapidly degraded by the fact that the total number of possible combinations is $C_{|E|}^{k-1}$, and generating all of them makes our algorithm work in super-polynomial time.

6.2. The *SplitNodes()* Algorithm.

A brute-force, optimal solution of in-place 2-VRP would consist of generating all two-set partitions of the set of leaves and computing the distance traveled by the servers during the process of serving the partitioned requests. Visiting all the leaves of the pruned tree is a sufficient criteria for serving all the requests, because it is impossible to visit all the leaves without also visiting all the internal nodes.

The *SplitNodes()* algorithm has two phases. First, it splits the pruned tree until it obtains a pre-established number of subtrees N . The splitting is done by creating a list of subtrees L that initially contains only the pruned tree. The algorithm picks the maximum-weight tree from L , splits it by removing all the edges connecting the root to its children, and adds the newly created subtrees to L . Second, *SplitNodes()* generates all two-set partitions of the list of subtrees L . For each partition of L , the algorithm assigns the subset L_i to the server s_i , and s_i serves the requests in L_i by traveling to the root of each subtree and by traversing the corresponding subtree in a depth-first manner. In the algorithm below, we used the notation T_v to denote the subtree of T that is rooted in the node v .

A crucial issue for our algorithm consists of visiting the subtrees in an efficient order. That is, we must minimize the distance traveled by a server while traveling from one subtree to the other one. A simple way to minimize this distance is the following: for each server s_i , we generate a new list of requests R_i that consists of all the roots of the subtrees in L_i , and we create a tree T_i obtained by pruning T_0 with respect to R_i . The server s_i traverses T_0 as if it would traverse T_i in a depth-first manner, and as soon as s_i

encounters a request-node from R_i , it serves all the requests in the corresponding subtree of T_0 .

It is easy to see that each server s_i performs a depth-first traversal of a subgraph of T_0 . It follows that each server travels a distance shorter than twice the weight of T_0 , and, consequently, the absolute performance ratio of *SplitNodes()* is $R_A \leq 2.0$.

SplitNodes(tree $T = (V, E)$, requests-list R , integer $MaxSubtrees$)

- let D_1, D_2 be the distances traveled by s_1, s_2
- let $root(T) = r$, and let L be the list of subtrees
- $T_0 = PruneTree(T, R)$
- FindPaths(r, T_0)
- ComputeDistances(r, T_0)
- GetSubtreeWeights(r, T_0)
- let $L = \{T_0\}$
- WHILE $|L| < MaxSubtrees$ DO
 - let T_v be the “heaviest” subtree in L
 - let v_1, v_2, \dots, v_k be the children of v
 - $L = (L - \{T_v\}) \cup \{T_{v_1}, \dots, T_{v_k}\}$
- FOR EACH partition $L = L_1 \cup L_2$ of L DO
 - let $L_1 = \{T_{v_1^1}, T_{v_2^1}, \dots, T_{v_{N_1}^1}\}$
 - let $L_2 = \{T_{v_1^2}, T_{v_2^2}, \dots, T_{v_{N_2}^2}\}$
 - $T_1 = PruneTree(T_0, \{v_1^1, v_2^1, \dots, v_{N_1}^1\})$
 - $T_2 = PruneTree(T_0, \{v_1^2, v_2^2, \dots, v_{N_2}^2\})$
 - $D_1 = TraverseSubtree(r, T_1)$
 - FOR $i = 1$ to N_1 DO
 - $D_1 = D_1 + TraverseSubtree(v_i^1, T_{v_i^1})$
 - $D_2 = TraverseSubtree(r, T_2)$
 - FOR $i = 1$ to N_2 DO
 - $D_2 = D_2 + TraverseSubtree(v_i^2, T_{v_i^2})$
 - IF $\max(D_1, D_2)$ is a new minimum THEN
 - update the minimum information
- return

GetSubtreeWeights (node n , tree $T = (V, E)$)

- $weight(n) = 0$
- FOR EACH child v of n DO
 - $weight(n) = weight(n) + TraverseSubtree(v, T)$
- return $weight(n)$

The tree-splitting phase takes $O(MaxSubtrees^2)$ to execute, while the second phase of the algorithm runs in $O((|V| + |R|) \times 2^{MaxSubtrees})$. Consequently, *SplitNodes()* also runs in $O((|V| + |R|) \times 2^{MaxSubtrees})$. Our algorithm can be easily modified to run for k VRP by generating all possible k -set partitions, but, again, increasing the value of k will rapidly turn the running time into a super-polynomial one.

6.3. The *BreakTree()* Algorithm.

This algorithm is based on the idea of splitting the pruned tree T_0 by creating a “border” between the areas of T_0 served by different servers (the “border” is defined as the path from $root$ to a leaf l of T_0). One server performs a depth-first traversal of the tree T_0 until it reaches the leaf l , and then returns directly to the root, while the other server traverses in a depth-first manner the remaining part of T_0 .

In order to check all possible borders, the *BreakTree()* algorithm starts by generating the list L of all leaves in T_0 . The leaves are added to L in the ordered in which they are reached during a depth-first traversal of T_0 . Once the list L is created, our algorithm successively breaks the pruned tree along the border defined by $root$ and each leaf in L . As both servers travel a distance shorter than twice the weight of T_0 , it follows that the absolute performance ratio of *BreakTree()* is $R_A \leq 2.0$.

BreakTree (tree $T = (V, E)$, requests-list R)

- let D_1, D_2 be the distances traveled by s_1, s_2
- let $r = root(T)$
- $T_0 = PruneTree(T, R)$
- FindPaths(r, T_0)
- ComputeDistances(r, T_0)
- compute the weight of T_0
- create the ordered list of leaves L
- FOR EACH leaf l DO
 - let l_{next} be the successor of l in L
 - $D_1 = BorderedTraverse(r, T_0, l, true) + dist(r, l)$
 - $D_2 = weight(T_0) + dist(r, l_{next}) -$
 - $BorderedTraverse(r, T_0, l_{next}, true)$
 - IF $\max(D_1, D_2)$ is a new minimum THEN
 - update the minimum information
- return

BorderedTraverse (node n , tree $T = (V, E)$, node b_{leaf} , boolean $flag$)

- $dist = 0$
- IF $flag == false$ THEN return 0
- IF $v == b_{leaf}$ THEN $flag = false$
- FOR EACH child v of n DO
 - $dist = dist + 2 \times dist(n, v) +$
 - $BorderedTraverse(v, T, b_{leaf}, flag)$
- return $dist$

By analyzing the algorithm above, we can see that *BorderedTraverse()* runs in $O(|E|)$, and, consequently, *BreakTree()* runs in $O(|R| + |E|^2)$. If we use $k - 1$ leaves to generate $k - 1$ borders, the *BreakTree()* algorithm partitions the tree into k areas that can be covered by k servers. However, modifying *BreakTree()* such that it solves k VRP is acceptable only for a few values of k , because even relatively small values of k make the algorithm run in super-polynomial time.

7. Approximation Algorithms for the common-source Problem

The algorithms above can be used almost unchanged to compute non-optimal solutions for the common-source problem. The only modification that we must perform is related to the manner in which we compute the distance traveled by each server. We shall replace the depth-first traversal approach with the simple summation of the double of the length of each path from the source to each destination. The above-mentioned summation can also be computed in $O(|E|)$, and the running times remain identical. Consequently, in Table 2 and Table 3 we will use the same names to denote slightly different variations of the *RemoveEdge()*, *SplitNome()*, and *BreakTree()* algorithms.

We introduce now two more algorithms for the common-source 2-VRP. *FirstFit()* is a greedy algorithm that can be also used for k -VRP without any increase of the running time, while the *ImprovedFirstFit()* algorithm generates slightly better solutions for 2-VRP, but for k VRP the running time increases rapidly for $k > 5$. Both *FirstFit()* and *ImprovedFirstFit()* split the set of requests between the two servers, and, consequently, their absolute performance ratio is also $R_A \leq 2.0$.

7.1. The *FirstFit()* Algorithm

This algorithm is based on a very simple idea. It starts by ordering the requests R in the descending order of the distance between the source and the destination, and continues by successively picking the *longest* unserved request and assigning it to the server that traveled the shortest distance.

As we can sort the requests in $O(|E| \times \log(|E|))$ by using heapsort or mergesort, it follows that *FirstFit()* runs in $O(|R| + |E| \times \log(|E|))$.

FirstFit (tree $T = (V, E)$, requests-list R)

- let r be the root of T
- let D_1, D_2 be the distances traveled s_1, s_2
- $T_0 = \text{PruneTree}(T, R)$
- $\text{FindPaths}(r, T_0)$
- $\text{ComputeDistances}(r, T_0)$
- sort R in the descending order of $\text{dist}(src, dest)$
- $D_1 = D_2 = 0$
- FOR EACH $(src, dest) \in R$ DO
 - IF $D_1 > D_2$ THEN $D_2 = D_2 + 2 \times \text{dist}(src, dest)$
 - ELSE $D_1 = D_1 + 2 \times \text{dist}(src, dest)$
- return

By replacing D_1 and D_2 with an array $D[k]$, we can easily use *FirstFit()* to solve k -VRP. All we have to do is to keep track of the maximum value stored in the array (let us assume that we store it in *MaxLoad*) and to assign the requests in

a circular manner (i.e., we keep assigning requests to s_i until it reaches at least *MaxLoad*, then update the value of *MaxLoad* and start the same process with s_{i+1}). It is important to note that the running time of *FirstFit()* remains the same, regardless of the number of existing servers.

7.2. The *ImprovedFirstFit()* Algorithm

ImprovedFirstFit() is extremely similar with *FirstFit()*. The only difference between the two algorithms consists of the fact that the improved version starts by a brute-force searching of the best possible partition of the heaviest N requests, where N is a fixed, pre-established number, and uses this pre-partition as a starting point for the partitioning process performed by *FirstFit()*.

ImprovedFirstFit(tree $T = (V, E)$, requests-list R , integer N)

- let r be the root of T
- let D_1, D_2 be the distances traveled by s_1, s_2
- $T_0 = \text{PruneTree}(T, R)$
- $\text{FindPaths}(r, T_0)$
- $\text{ComputeDistances}(r, T_0)$
- sort R in the descending order of $\text{dist}(src, dest)$
- let R_1 contain the first N requests from R
- let $R_2 = R - R_1$
- FOR EACH partition $R_1 = P_1 \cup P_2$ DO
 - $D_1 = D_2 = 0$
 - FOR EACH $(src, dest) \in P_1$ DO
 - $D_1 = D_1 + 2 \times \text{dist}(src, dest)$
 - FOR EACH $(src, dest) \in P_2$ DO
 - $D_2 = D_2 + 2 \times \text{dist}(src, dest)$
 - IF $\max(D_1, D_2)$ is a new minimum THEN
 - update the minimum information
- let D_1, D_2 be the best values obtained above
- FOR EACH $(src, dest) \in R_2$ DO
 - IF $D_1 > D_2$ THEN $D_2 = D_2 + 2 \times \text{dist}(src, dest)$
 - ELSE $D_1 = D_1 + 2 \times \text{dist}(src, dest)$
- return

As the pre-partitioning process takes $O(2^N)$, the algorithm above runs in $O(|R| + |E| \times \log(|E|) + 2^N)$. It follows that even for significantly large values of N (e.g., $\log(|E|^2)$ or $\log(|E|^3)$), the running time of the algorithm remains quite acceptable (e.g., $O(|R| + |E|^2)$ or $O(|R| + |E|^3)$). On the other side, by increasing the number of servers in *ImprovedFirstFit()*, we will dramatically slow down the algorithm because of the time necessary to generate all possible k -set partitions of the “heaviest” N requests.

	“Average” R_A				“Worst-Case” R_A			
	nb, w	nb, uw	b, w	b, uw	nb, w	nb, uw	b, w	b, uw
RemoveEdge	1.062	1.066	1.072	1.059	1.296	1.335	1.261	1.219
SplitNodes	1.016	1.024	1.016	1.015	1.282	1.266	1.102	1.099
BreakTree	1.052	1.055	1.037	1.009	1.219	1.240	1.242	1.092

Table 2. “Average” and “Worst-Case” R_A for in-place 2-VRP

	“Average” R_A				“Worst-Case” R_A			
	nb, w	nb, uw	b, w	b, uw	nb, w	nb, uw	b, w	b, uw
RemoveEdge	1.121	1.131	1.123	1.121	1.399	1.457	1.311	1.304
SplitNodes	1.076	1.065	1.006	1.011	1.678	1.651	1.047	1.091
FirstFit	1.016	1.009	1.011	1.012	1.056	1.057	1.039	1.065
ImprovedFirstFit	1.017	1.008	1.009	1.014	1.053	1.047	1.041	1.065

Table 3. “Average” and “Worst-Case” R_A for common-source 2-VRP

8. Approximation Algorithms for the common-destination Problem

The common-source and common-destination problems are extremely similar in nature. In both problems, a server s that serves a request $(src, dest)$ travels exactly twice the length of the path from src to $dest$. Consequently, all the approximation algorithms presented in the previous section can be also used for the common-destination problem.

9. Experimental Results

In this section, we present the “average” R_A and the “worst-case” R_A obtained by running our approximation algorithms for sets of random trees. The “average” R_A results represent the average of the R_A values computed for 4000 random trees (we generated trees with 25, 30, 35, and 40 nodes), while the “worst-case” R_A represent the largest values of R_A encountered for the same set of 4000 trees. All the algorithms were implemented in C, and the programs were executed on a Sparc 5 station running the Solaris 4.2 operating system.

In Table 2 we present our result for the in-place 2-VRP problem, while in Table 3 we show the results for the common-source 2-VRP, which are identical with the ones of common-destination 2-VRP. Each column in a table contains the results for a different type of underlying graph: “non-binary, weighted trees” (nb, w), “non-binary, unweighted trees” (nb, uw), “binary, weighted trees” (b, w), respectively “binary, unweighted trees” (b, uw).

There are several interesting comments about our results. For the in-place problem, *SplitNodes()* has the best

“average R_A ” for three out of the four types of underlying graphs. However, the naive *BreakTree()*, which has the best “average” for binary, unweighted trees, obtains far better “worst-case R_A ” values in comparison with the other two algorithms.

By analyzing the data for the common-source problem, we can see other interesting results. First, for small values N , the “average R_A ” values of *ImprovedFirstFit()* are just slightly better than the ones of *FirstFit()*. Furthermore, for the “worst-case R_A ”, *FirstFit()* outperforms *ImprovedFirstFit()* on binary, weighted trees and equals its performances on binary, unweighted trees. Second, excepting the “average R_A ” of *SplitNodes()* for binary, weighted trees, the two variants of *FirstFit()* behave far better than *RemoveEdge()* and *SplitNodes()*.

10. Conclusions

In this paper, we introduced several NP-complete variants of the Vehicle Routing Problem in trees. We mainly focused on the 2-vehicle routing problem, and we presented three approximation algorithms for the in-place variants, and five algorithms for the common-source and common-destination problems. All these algorithms are extremely fast, and the experimental results showed excellent approximation ratios. We also considered possible generalizations for k -VRP, but excepting the *FirstFit()* algorithm, the other ones do not scale well to a number of servers larger than 5.

References

- [1] M. Atallah and S. Kosajaru. Efficient solutions to some

- transportation problems with application to minimizing robot arm travel. *SIAM Journal of Computing*, 17:849–869, 1988.
- [2] I. Averbakh and O. Berman. Sales-delivery-man problems on treelike networks. *Networks*, 25:45–58, 1995.
- [3] M. Chobrak and L. Larmore. On fast algorithms for two servers. *Journal of Algorithms*, 12:607–614, 1991.
- [4] S. Cook. The complexity of theorem proving procedures. *Proceedings of the Third ACM Symposium on Theory of Computing*, pages 151–158, 1971.
- [5] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
- [6] G. Frederickson. A note on the complexity of the simple transportation problem. *SIAM Journal of Computation*, 22(1):57–61, 1993.
- [7] G. Frederickson and D. Guan. *Non-preemptive Ensemble Motion Planning on a Tree*. Technical Report CSD-TR-864, Department of Computer Science, Purdue University, 1992.
- [8] G. Frederickson and D. Guan. Preemptive ensemble motion planning on a tree. *SIAM Journal of Computing*, 21(6):1130–1152, 1992.
- [9] G. Frederickson, M. Hecht, and C. Kim. Approximation algorithms for some routing problems. *SIAM Journal of Computation*, 7(2):178–193, 1978.
- [10] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, CA, 1979.
- [11] Y. Karuno, H. Nagamochi, and T. Ibaraki. Vehicle scheduling on a tree with release and handling times. *Proceedings of the International Symposium on Algorithms and Algebraic Computing (ISAAC), Springer-Verlag Lecture Notes in Computer Science 762*, pages 486–494, 1993.
- [12] W. Klostermeyer. The k-server problem in dynamically orientable graphs. *Forthcoming*, 1997.
- [13] W. Klostermeyer. Paths problems in dynamically orientable graphs. *Forthcoming*, 1997.
- [14] W. Klostermeyer. The very offline two-server problem. *Forthcoming*, 1997.
- [15] R. Miller and J. Thatcher. *Complexity of Computer Computations*. Plenum Press, New York, NY, 1972.
- [16] I. Muslea. *The k-Vehicle Routing Problem in Trees*. M.S. thesis, Department of Statistics and Computer Science, West Virginia University, 1996.
- [17] M. Solomon. Algorithms for the vehicle routing and scheduling problem with time window constraints. *Operations Research*, 35(2):254–265, 1987.
- [18] M. Solomon and J. Desrosiers. Time window constrained routing and scheduling problems: A survey. *Transportation Sci*, 22:1–13, 1988.