

MULTI-VEHICLE ONE-TO-ONE PICKUP AND DELIVERY
PROBLEM WITH SPLIT LOADS

Mustafa Şahin

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of the requirements for the degree of
Master of Science

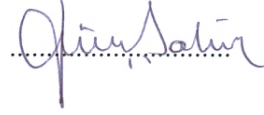
Sabanci University

August, 2011

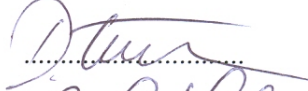
MULTI-VEHICLE ONE-TO-ONE PICKUP AND DELIVERY
PROBLEM WITH SPLIT LOADS

Approved by:

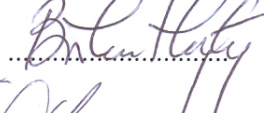
Assist. Prof. Dr. Güvenç Şahin
(Thesis Supervisor)



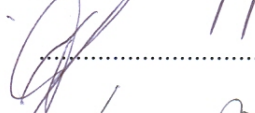
Assist. Prof. Dr. Dilek Tüzün Aksu



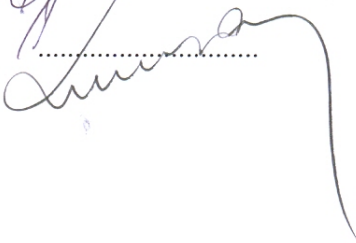
Assoc. Prof. Dr. Bülent Çatay



Assoc. Prof. Dr. Temel Öncan



Prof. Dr. Gündüz Ulusoy



Date of Approval: 08/09/2011

to my family...

Acknowledgements

I would like to convey my sincerest gratitude towards Dr. Güvenç Şahin, the thesis supervisor with an 'S' on his chest, for his unyielding support and guidance, for looking out for me, for believing in me and countless other things that I am not mentioning here.

I wish to thank Dr. Dilek Tüzün Aksu and Dr. Temel Öncan for their invaluable feedback, guidance and support, my degree would not have been possible without them.

I wish to thank the Scientific and Technological Research Council of Turkey for financial support.

I owe a great deal to my friends who helped me through thick and thin. Halit, my flatmate, my high school buddy, we did so much together, thanks for helping me in every aspect of my life and for answering my stupid questions about computer stuff. Gizem, thanks for helping me when I needed related or unrelated to this work, for letting me borrow your notes, your desk and so many other things, thanks simply for being you. Arda, thanks for being my movie mate, tennis partner, project partner and listening to my idiotic rants about most trivial things, it was great to have you on the other side of the net. Emrah, the president, thanks for simply being in my life, you have no idea how great it feels to have someone on whom I can thoroughly rely. Eren and Serdar, thank you for calling me from the States in spite of the outrageous phone charges when I needed to hear your voices. Sezer, Yasin, Gökтуğ and Ahmet, though we are unable to meet often, I thank you guys for being with me through the best years of my life, memory of those years gives me fuel during hard times. Suat, Kerem, Emin and Çağlar, without you guys, I would never step on to the sunlight, thanks for taking me outside of the house every once in a while and staying in with me when I didn't want to leave.

Finally, I want to thank my family, to whom I owe everything. Thanks dad for making me the man I am today by setting an example and helping me trying to get there. Thank you mom for believing in me when even I didn't believe in myself, I am proud to be your son. My brother, thank you for stocking the fridge with my favorites for my home visits. My cousin Dr. Seçkin Sunal, thank you for being a safe harbor for me in this god forsaken city, you have no idea how great it feels to be near family, thank you for being the best person I know. I thank everyone who has ever loved me, thank you for making everything possible for me.

© Mustafa Şahin 2011

All Rights Reserved

Parçalanabilir Yüklü Toplama ve Dağıtma Araç Rotalama Problemi

Mustafa Şahin

Endüstri Mühendisliği, Yüksek Lisans Tezi, 2011

Tez Danışmanı: Güvenç Şahin

Anahtar Kelimeler: Toplama ve Dağıtma, Araç Rotalama, Parçalanabilir Yük, Tabu Arama, Benzetimli Tavlama.

Özet

Bu çalışmada Parçalanabilir Yüklü Toplama ve Dağıtma Araç Rotalama Problemi (MPDPSL) ele alınmıştır. Bu problem, bire bir Toplama ve Dağıtma Probleminin bir yükün farklı araçlar tarafından ya da bir araç tarafından birden çok seferde sağlanabildiği bir uzantısıdır. Uygulamada, yükün fiziksel olarak parçalanabildiği alanlarda, parçalanabilir dağıtım geçerli bir seçenek olarak 3. taraf lojistik işletmelerinin kurye servislerinde kullanılmaktadır. Aynı zamanda bu problemin, yüklerin sabit bir maliyet tarafından dışarıdan bir firma tarafından taşınabildiği (MPDPSL-O) ve rotaların depo olmadan döngüsel olduğu (MPDPSL-C) iki varyantı ele alınmıştır. Problemin ve iki varyantının çözümünde tabu arama ve benzetimli tavlamanın güçlü yönlerini buluşturan bir sezgisel algoritma geliştirilmiştir. Yazında yer alan bir problem kümesi üzerinde yapılan deneyler sonucunda, sezgiselin makul sürelerde iyi sonuçlar verdiği saptanmıştır. Yazındaki başka bir problem kümesi için ise ilk sonuçlar ortaya konmuştur ve parçalanabilir yükün dağıtım ağının yapısına bağlı olarak sağladığı faydalar incelenmiştir. Çeşitli maliyet yapıları altında dışarıdan teminin potansiyel faydalarını incelemek amacıyla MPDPSL ve MPDPSL-O karşılaştırılmıştır. Son olarak, MPDPSL-C için gerçek bir vaka çözülmüştür.

MULTI-VEHICLE ONE-TO-ONE PICKUP AND DELIVERY PROBLEM WITH SPLIT LOADS

Mustafa Şahin

Industrial Engineering, Master's Thesis, 2011

Thesis Supervisor: Asst. Prof. Dr. Güvenç Şahin

Keywords: Pickup and Delivery, Vehicle Routing, Split Loads, Tabu Search, Simulated Annealing.

Abstract

In this study, we consider the Multi-vehicle One-to-one Pickup and Delivery Problem with Split Loads (MPDPSL). This problem is a generalization of the one-to-one Pickup and Delivery Problem (PDP) where each load can be served by multiple vehicles as well as multiple stops by the same vehicle. In practice, split deliveries is a viable option in many settings where the load can be physically split, such as courier services of third party logistics operators. We also consider two other variants of the problem where it is possible to outsource the pickup and delivery requests for a fixed charge (MPDPSL-O) and where the routes are cyclic without depot (MPDPSL-C). We propose an efficient heuristic that combines the strengths of Tabu Search and Simulated Annealing for the solution of MPDPSL and its variants. Results from experiments on a problem set in the literature indicate that the heuristic is capable of producing good quality solutions in reasonable time, we present first results on another problem set in the literature and discuss the merits of load splitting with respect to the network distribution. We compare the results of MPDPSL and MPDPSL-O in order to illustrate the potential benefits of outsourcing under various outsourcing cost schemes. Finally, we present a solution for a real life case of MPDPSL-C.

Contents

1	Introduction	1
2	Mathematical Model for MPDPSL	4
3	MPDPSL Heuristic	8
3.1	Initial Solution Heuristic	9
3.2	Insert/Split Phase	10
3.3	Intra-Route Phase	13
3.4	Block Insert Phase	13
3.5	Block Swap Phase	13
3.6	Tabu Embedded Simulated Annealing (TESA) Algorithm	14
3.7	Computational Results	17
3.7.1	Parameter Tuning for TESA Algorithm	17
3.7.2	Computational Experiments on Test Problems	22
3.7.3	Analysis of the Benefit Obtained by Split Loads	23
3.7.4	Comparative Analysis of Algorithm Performance	26
4	MPDPSL with Outsourcing Option	30
4.1	Modified TESA Algorithm for MPDPSL-O	31
4.2	Computational Results	32
4.2.1	Parametric Settings for MPDPSL-O	33
4.2.2	Benchmarking MPDPSL-O Solutions with MPDPSL Solutions	34
5	MPDPSL with Cycles without Depot	37
5.1	Modified TESA Algorithm for MPDPSL-C	40
5.2	Computational Results	41
6	Concluding Remarks	43

Appendices	48
A Algorithm Description and Computational Complexity Analysis	48
A.1 Notation	48
A.2 Software Architecture and Data Structure	48
A.3 Insert/Split Phase	49
A.4 Block Insert Phase	52
A.5 Block Swap Operator	53
B Results of Ropke and Pisinger Problem Instances	54
B.1 Split vs. No-split Solutions	54
B.2 Computational Times	56

List of Figures

3.1	Improvement in route length versus CPU time for different values of N_{iter} .	18
3.2	Improvement in route length versus CPU time for different values of N_{pair} .	19
3.3	Improvement in route length versus CPU time for different values of N_{move} .	19
3.4	Combined effect of N_{iter} and N_{pair} on the improvement in route length.	20
3.5	Combined effect of N_{iter} and N_{move} on the improvement in route length.	20
3.6	Combined effect of N_{move} and N_{pair} on the improvement in route length.	21
3.7	The pattern of route length improvement through the restarts of the algorithm.	22
3.8	Effect of the load range on the improvement in route length.	25
3.9	Effect of the load range and spatial distribution of points on the improvement in route length.	26
5.1	An example for a cyclic route with four ports	39
5.2	MPDPSL-C representation of the cycle in Figure 5.1	40
5.3	A cyclic route from the real life instance of MPDPSL-C	42

List of Tables

3.1	Improvement in the route length obtained by split loads for problems in Nowak et al. [10] for a load size range of [0.51-0.6] truckload.	24
3.2	Results for the single vehicle problems with 75 requests from Nowak et al. [10].	28
3.3	Results for the single vehicle problems with 100 requests from Nowak et al. [10].	29
3.4	Results for the single vehicle problems with 125 requests from Nowak et al. [10].	29
4.1	Results for 0.25-1 load range with <i>triangular</i> distance scheme from Ropke and Pisinger. [13].	34
4.2	Results for 0.25-1 load range with <i>direct</i> distance scheme from Ropke and Pisinger. [13].	34
4.3	Results for 0.51-0.60 load range with <i>triangular</i> distance scheme from Ropke and Pisinger. [13].	34
4.4	Results for 0.51-0.60 load range with <i>direct</i> distance scheme from Ropke and Pisinger. [13].	35
5.1	p-d requests for the cycle in Figure 5.1	39
5.2	MPDPSL-C representation of the cycle in Figure 5.1	40
5.3	Results from an MPDPSL-C instance	42
B.1	Results of the problems in Ropke and Pisinger [13] with 50 nodes comparing the tour length of split solutions versus no-split solutions.	54
B.2	Results of the problems in Ropke and Pisinger [13] with 100 nodes comparing the tour length of split solutions versus no-split solutions.	55
B.3	Results of the problems in Ropke and Pisinger [13] with 250 nodes comparing the tour length of split solutions versus no-split solutions.	55

B.4	Results of the problems in Ropke and Pisinger [13] with 500 nodes comparing the tour length of split solutions versus no-split solutions.	55
B.5	CPU time per restart (in seconds) to solve the problems when the load size range is 0.25-1.	56
B.6	CPU time per restart (in seconds) to solve the problems when the load size range is 0.51-0.60.	56

Chapter 1

Introduction

The Pickup and Delivery Problem has attracted the interest of various researchers in the last three decades (see e.g. recent surveys by Cordeau et al.[4], Gribkovskaia and Laporte [7]). PDP consists of finding the least cost route of a single vehicle which picks up a load from an origin and delivers it to its destination. If each origin is associated with a single destination, making up a pickup-delivery (p-d) pair, the problem is called one-to-one PDP. This version of PDP differs from the other two variants in the literature: The many-to-many PDP where a commodity may be picked up at one of many origins and then delivered to one of many destinations, and the one-to-many-to-one PDP where all loads to be picked-up and delivered originate at a common depot (Cordeau et al. [4]). In a conventional PDP setting, each p-d pair is visited by a single vehicle. We study the version where the load of a p-d pair can be split between multiple vehicles and/or multiple stops of the same vehicle. The single vehicle version of this problem, namely the Pickup and Delivery Problem with Split Loads (PDPSL) was first introduced by Nowak et al. [10].

The Multi-vehicle One-to-one Pickup and Delivery Problem with Split Loads (MPDPSL), which is a generalization of PDPSL, has not been widely studied in the literature; though the possibility of load splitting exists in many PDP settings. Nowak et al. [10] describe the less-than-truckload service of a third-party logistics company where load splitting results in significant benefits. Similar savings might be achieved in other areas where loads can be split among multiple vehicles, such as bulk product transportation by ship, where each load is already packaged into multiple containers, or courier services that deliver multiple packages between the same origin-destination pair. As long as loads can be physically split between vehicles, MPDPSL can lead to savings over the classical

PDP by reducing the unused vehicle capacity.

Nowak et al. [10] have shown that the optimal load size for splitting is just above one half of a truckload and demonstrated the relation between the benefit of split loads and the problem characteristics on the third-party logistics case study. The benefit of split loads and the problem characteristics that are affected most by load splitting have been experimentally analyzed in their subsequent work (Nowak et al. [11]). They have empirically observed that up to 30% savings in route cost can be achieved when the demands are distributed between 51% and 60% of the truckload (vehicle capacity). Other factors that increase the benefit of split loads are the number of loads available at a common location for pickup or delivery and the average distance from an origin to a destination relative to the distance from origin to origin and destination to destination. They have also observed that an increase in these two factors result in an increase in the benefit of split loads.

The split load case is also considered within the VRP context. The well-known Split Delivery Vehicle Routing Problem (SDVRP) tries to find a set of minimum cost routes for a fleet of capacitated homogeneous vehicles available to serve a set of customers. Each customer can be visited more than once and the demand of each customer may be greater than the vehicle capacity. The earliest papers on SDVRP trace back to the work by Dror and Trudeau [6]. In a more recent work, Archetti et al. [1] provide an extensive empirical analysis on the benefit of splitting deliveries for the classical VRP. For a recent survey on SDVRP, we refer to Archetti and Speranza [2].

To the best of our knowledge, there is no other work in the literature that deals with the multi-vehicle extension of PDPSL. The motivation of this work is to present MPDPSL, and then to introduce our Tabu Embedded Simulated Annealing (TESA) algorithm specially tailored for this problem. Computational experiments are performed on both PDPSL test problems and split load versions of PDP instances from the literature. We also study two variants of MPDPSL:

- in MPDPSL with outsourcing option, it is possible to outsource any p-d request for a fixed charge,
- in MPDPSL with cyclic routes, the routes are cyclic and there are no vehicle depots.

Our main contributions in this work are as follows:

- we introduce a generalized version of PDPSL with multiple vehicles;
- a metaheuristic algorithm is designed to solve MPDPSL;

- we perform computational experiments to
 - show the efficiency and effectiveness of our algorithm,
 - analyze the benefit of split loads for the multi-vehicle problem with different network structures;
- we report the first results on a set of MPDPSL test problems;
- we introduce two variants of the MPDPSL and tweak our existing algorithm for these problems;
- we perform computational experiments to
 - analyze the benefit of the outsourcing option, and
 - introduce results for a real life case of cyclic routes.

The remainder of this work is organized as follows. In Chapter 2, we introduce a mixed integer programming formulation for MPDPSL. In Chapter 3, we discuss the details of TESA heuristic algorithm presented for MPDPSL. We present the results of computational experiments in Section 3.7. Then we introduce the variants of MPDPSL in Chapter 4 and Chapter 5 and the respective computational analysis in Section 4.2 and Section 5.2. Finally, we close with concluding remarks in Chapter 6.

Chapter 2

Mathematical Model for MPDPSL

MPDPSL is defined on a directed graph $G = (V, A)$. The vertex set is partitioned as $V = \{P, D, \{0, 2n + 1\}\}$. For a given set of n pickup-delivery pairs, $P = \{1, 2, \dots, n\}$ is the set of pickup vertices and $D = \{n + 1, \dots, 2n\}$ is the set of delivery vertices where i and $n + i$ represent the pickup and delivery vertices for load i respectively; $\{0, 2n + 1\}$ includes two copies of the depot location. The set of arcs are defined as follows $A = \{(i, j) : i = 0, j \in P\} \cup \{(i, j) \in P \cup D, i \neq j, i \neq n + j\} \cup \{(i, j) : i \in D, j = 2n + 1\}$. $K = \{1, 2, \dots, m\}$ denotes the set of available vehicles. For each vertex $i \in P$, q_i denotes the p-d request quantity where $q_{i+n} = -q_i$. Each vehicle $k \in K$ has a capacity of Q . We assume that the load of any p-d pair can be provided by a single vehicle, i.e. $q_i \leq Q, \forall i \in P$. d_{ij} is the travel distance associated with arc $(i, j) \in A$, and L is the maximum travel distance of vehicle route. $x_{ij}^k \in \{0, 1\}, \forall i, j \in V, \forall k \in K$ denotes a binary decision variable: $x_{ij}^k = 1$ if arc (i, j) is used by vehicle k ; 0, otherwise. $y_i^k \in [0, 1], \forall i \in V, \forall k \in K$ denotes the fraction of demand of p-d pair $(i, n + i)$ satisfied by vehicle k . l_i^k denotes the load of vehicle $k \in K$ after visiting vertex $i \in V$, and $l_0^k = l_{2n+1}^k = 0$. u_i^k and $w_i^k, \forall i \in V, \forall k \in K$ are supporting decision variables used to calculate the precedence and distance traveled respectively. The resulting mixed integer programming model for MPDPSL is as follows:

$$\text{Minimize } \sum_{k \in K} \sum_{i \in V} \sum_{j \in V} d_{ij} x_{ij}^k \quad (2.1)$$

$$\text{subject to } \sum_{k \in K} y_i^k = 1 \quad \forall i \in P \cup D \quad (2.2)$$

$$\sum_{j \in P \cup D} x_{ij}^k = \sum_{j \in P \cup D} x_{ji}^k \quad \forall i \in P \cup D, \forall k \in K \quad (2.3)$$

$$\sum_{i \in P \cup \{2n+1\}} x_{0i}^k = 1 \quad \forall k \in K \quad (2.4)$$

$$\sum_{i \in D \cup \{0\}} x_{i,2n+1}^k = 1 \quad \forall k \in K \quad (2.5)$$

$$y_i^k \leq \sum_{j \in P \cup D} x_{ij}^k \leq M y_i^k \quad \forall i \in P \cup D, \forall k \in K \quad (2.6)$$

$$y_i^k \leq \sum_{j \in P \cup D} x_{ji}^k \leq M y_i^k \quad \forall i \in P \cup D, \forall k \in K \quad (2.7)$$

$$\sum_{j \in P \cup D} x_{ij}^k = \sum_{j \in V - \{0\}} x_{i+n,j}^k \quad \forall i \in P, \forall k \in K \quad (2.8)$$

$$\sum_{j \in V - \{2n+1\}} x_{ji}^k = \sum_{j \in P \cup D} x_{j,i+n}^k \quad \forall i \in P, \forall k \in K \quad (2.9)$$

$$l_i^k - l_j^k + Q x_{ij}^k + y_j^k q_j \leq Q \quad \forall i \in P, \forall j \in P \cup D, \quad \forall k \in K \quad (2.10)$$

$$y_i^k q_i \leq l_i^k \leq Q \sum_{j \in P \cup D} x_{ij}^k \quad \forall i \in P, \forall k \in K \quad (2.11)$$

$$0 \leq l_i^k \leq (Q + q_i) \sum_{j \in P \cup D} x_{ij}^k \quad \forall i \in D, \forall k \in K \quad (2.12)$$

$$l_0^k = l_{2n+1}^k = 0 \quad \forall k \in K \quad (2.13)$$

$$u_i^k - u_j^k + (M-1)x_{ij}^k + (M-3)x_{ji}^k \leq M-2 \quad \forall i \in V - \{2n+1\}, \quad \forall j \in V - \{0\}, \forall k \in K \quad (2.14)$$

$$u_i^k \leq (M-1) - (M-3)x_{0i}^k - \sum_{j \in P \cup D} x_{ij}^k \quad \forall i \in P \cup D, \forall k \in K \quad (2.15)$$

$$u_i^k \leq u_{i+n}^k - 1 \quad \forall i \in P, \forall k \in K \quad (2.16)$$

$$w_i^k - w_j^k + (L + d_{ij})x_{ij}^k + (L - d_{ji})x_{ji}^k \leq L \quad \forall i \in V, \forall j \in V \forall k \in K \quad (2.17)$$

$$w_0^k = 0 \quad \forall k \in K \quad (2.18)$$

$$w_{2n+1}^k \leq L \quad \forall k \in K \quad (2.19)$$

$$x_{ij}^k \in \{0, 1\} \quad \forall i \in V, \forall j \in V, \forall k \in K \quad (2.20)$$

$$y_i^k \in [0, 1] \quad \forall i \in P \cup D, \forall k \in K \quad (2.21)$$

$$0 \leq l_i^k \leq Q \quad \forall i \in V, \forall k \in K \quad (2.22)$$

$$u_i^k \in [0, 2n + 1] \quad \forall i \in V, \forall k \in K \quad (2.23)$$

$$w_i^k \geq 0 \quad \forall i \in V, \forall k \in K \quad (2.24)$$

The objective function, (2.1), calculates the total distance traveled by all vehicles. Constraint (2.2) ensures that the demand of a p-d pair is fully satisfied. With constraint (2.3), when an arc enters a vertex, another arc must leave that vertex. Constraints (2.4) and (2.5) ensure that an arc leaves and enters the depot for all vehicles. By constraints (2.6) and (2.7), a vertex is visited if a fraction or all of its demand is picked up or delivered. Constraints (2.8) and (2.9) ensure that an arc must enter/leave its corresponding delivery point if an arc enters/leaves a pickup point. Constraints (2.10) - (2.13) calculate the load of the vehicle ensuring that the load is below the capacity at all times and a vehicle is empty while leaving and entering the depot. Constraints (2.14) - (2.16) represent the precedence relations: Constraint (2.14) establishes the order between two vertices when one vertex is visited right before the other. Constraint (2.15) assigns the vertex visited right after depot with the smallest order. Constraint (2.16) ensures that a delivery point is not visited before its corresponding pickup point. Constraint (2.17) calculates the distance traveled up to each vertex by aggregating the distances from depot. Constraint (2.18) sets the distance traveled to zero at the beginning of the route and constraint (2.19) ensures that the total distance traveled at the route is less than the maximum distance allowed L . Constraints (2.20) - (2.24) are the domain constraints for the decision variables.

MPDPSL relates to PDP to a great extent since both models have *coupling* and *precedence* constraints. A coupling constraint ensures that a delivery vertex is visited when its corresponding pickup vertex is visited and vice versa. A precedence constraint ensures that a delivery vertex is not visited before its corresponding pickup request. PDP is known to be NP-Hard (Lenstra and Kan [8]) and MPDPSL contains PDP as a special case, hence, MPDPSL is also NP-Hard. Therefore, we have no hope of solving large scale instances

of MPDPSL to optimality in reasonable time as the problem becomes intractable very quickly. In the proposed mixed integer programming model, splitting a p-d request in the same route is not allowed since allowing it would require a more complex model and would increase the computational time. Limiting the maximum number of splits would also lessen the computational burden; however, even after limiting the maximum number of splits to two, we are not able solve instances with more than 6-7 pairs in reasonable time. To sum up, MPDPSL is even more complicated and more challenging to solve to optimality than VRP and PDP, therefore, we recourse to heuristics.

Chapter 3

MPDPSL Heuristic

We propose a Tabu Embedded Simulated Annealing (TESA) heuristic for MPDPSL. The heuristic utilizes features of both tabu search and simulated annealing to search the solution space efficiently. In the literature, tabu search and simulated annealing methods have been used together in metaheuristic algorithms. Li et al. [9] and Thangiah et al. [14] use simulated annealing to obtain non-tabu neighbor solutions for the Vehicle Routing Problem with Time Windows (VRPTW). Osman [12] employs this approach to solve the Generalized Assignment Problem. Our approach follows the footsteps of Li et al. [9] and Thangiah et al. [14]; we use a simulated annealing-like procedure to select among a set of possible elicited moves while searching the neighborhood of a current solution. In particular, in order to jump to a neighboring solution, we do not necessarily choose to apply the best possible move; we consider a list of good moves, and use a randomized procedure to select a move from this list. The simulated annealing procedure is employed to direct the randomization scheme.

The heuristic starts by creating an initial solution using a variant of the savings heuristic by Clarke and Wright [3]. This solution is then improved by searching neighboring solutions through a variety of moves, including insertion of a p-d pair into a different route, split of a p-d pair between routes as well as insertion and swap of route segment(s) that consist of multiple p-d pairs. The search neighborhoods to be used in a metaheuristic are very critical to the efficiency of the algorithm. This is especially true for an MPDPSL heuristic where the neighborhoods are potentially very large due to many possible ways of splitting a load between available routes.

The heuristic algorithm subsequently goes through a series of improvement phases following the initialization with a feasible solution. In each phase, the solution is im-

proved by searching new solutions in the corresponding neighborhood of the particular phase. In the remainder of this chapter, we first give the details of the initial solution heuristic and the improvement phases before we discuss the overall algorithm.

While discussing the details of the algorithm, we represent a *solution* $S = \{R_1, R_2, \dots, R_{|S|}\}$ as a collection of routes R that consist of pickup and delivery stops. We prefer to use the term *stop* instead of pickup and delivery points since we can split a p-d request. Therefore, we refer to each pickup and delivery point, split or otherwise, as a stop. A p-d pair consists of a pickup stop and the corresponding delivery stop with indices p and d . $s(x)$ denotes the savings function whereas $c(x)$ is the cost function. For instance, $s(p-d)$ denotes the decrease in route length obtained by removing the pair p-d from its current route.

$$s(p-d) = d_{p'p} + d_{pp''} - d_{p'p''} + d_{d'd} + d_{dd''} - d_{d'd''}$$

where p' and d' denote the stops before p and d , p'' and d'' denote the stops after p and d respectively.

$c(R)$ denotes the cost of route R and $c(S)$ denotes the cost of the solution S . We define *block* as a route segment that starts and ends with an empty vehicle. While there may be several blocks within a route, a route may also consist of a single block. B denotes a block and ℓ^B denotes the location where B is inserted.

3.1 Initial Solution Heuristic

The heuristic algorithm starts by creating an initial solution. This initial solution, which is based on the savings algorithm by Clarke and Wright [3], does not contain any split loads. In parallel with the original savings algorithm, we first create a solution where each p-d pair is initially served by a separate vehicle route of the form $(0, p_i, d_i, 0)$. We then compute a savings value for every pair of pickup point p_i and delivery point d_i such that

$$s(d_i, p_j) = d_{d_i 0} + d_{0 p_j} - d_{d_i p_j} \quad \forall i \neq j \in P$$

The savings value is the difference in total route length when the routes that serve the two points d_i and p_j are combined into a single route. Next, we sort the pairs in non-increasing order of their savings. Starting from the first pair on the list, we merge the routes that visit the pairs with positive savings while ensuring the feasibility of the resulting routes with respect to vehicle capacity and the maximum travel distance. Finally, we carry out an improvement step where the pickup point p_i and delivery point d_i are moved forward and

backward respectively in the merged route provided that such a move results in further savings.

3.2 Insert/Split Phase

We improve a given solution through subsequent moves in the insert/split neighborhood. A neighboring solution in this neighborhood is constructed by removing a p-d pair from its current route and either inserting the entire load into a different route or splitting it between two routes. Each iteration corresponds to a feasible move that does not violate the precedence, vehicle capacity and route length constraints. We limit the number of splits for a given p-d pair to a maximum of k and since there are infinitely many ways to split a load between two routes, we use up the excess capacity in one route and split the remaining demand to the other route. In a feasible solution, there are at most $2nk$ stops. Thus, for a given p-d pair, there are $O(n^2)$ ways of inserting the pair into two positions in another route, and $O(n^4)$ ways of splitting the pair between two other routes. As a result, the computational complexity of evaluating the entire insert/split neighborhood for a particular p-d pair and identifying the best insert/split move among all p-d pairs are in $O(n^4)$ and $O(n^5)$, respectively.

Since computational complexity of an insert/split move for a particular p-d pair is very high, it is quite time consuming to evaluate all p-d pairs at every iteration. In order to avoid this excessive computational burden, we evaluate the insert/split neighborhood using a binary heap implementation to improve the average computational performance. While the worst-case behavior of the binary heap implementation is the same ($O(n^5)$), our computational experience indicates that the average running time is only $O(n^3 \log n)$, which is significantly better than the original implementation. (A detailed description of the binary heap implementation and its computational complexity can be found in Appendix A.)

In order to select the p-d pair to be inserted/split in each iteration, we first evaluate the insert/split neighborhood for all p-d pairs to determine the best N_{move} non-tabu moves that result in the smallest change in total route length (ΔC) value for each pair as a result of the move. Note that a positive ΔC value corresponds to an increase in the route length.

The tabu structure in this algorithm consists of four components: the route length, the number of stops and the number of vehicles in the visited solution, as well as the p-d pair inserted/split to create that solution. The reason for using such a complex tabu structure is due to the special property of the test problems by Nowak et al.[10]. In

these test problems, all p-d pairs are generated from a limited set of pickup and delivery points. Since there are many nodes that share the same pickup and delivery coordinates, conventional tabu structures that keep track of involved node(s) are rendered ineffective for these test problems. Using the multi-dimensional tabu structure that keeps track of the features of the visited solution (in addition to the move applied to reach that solution) allows the algorithm to identify previously visited solutions correctly and avoids visiting them repeatedly later on.

Once we create a list of the top N_{move} non-tabu moves with the lowest ΔC values among all feasible insert and split moves for every p-d pair, we employ a simulated annealing based selection criterion to select one candidate move for each p-d pair. In a conventional simulated annealing algorithm, a move is selected with a probability $e^{-\Delta C/temp}$, where $temp$ denotes the current temperature of the simulated-annealing-like randomized selection procedure. The average value of ΔC may fluctuate significantly between p-d pairs. Therefore, the acceptance probability of a move with an average ΔC value also changes widely. To prevent this, we use the following approach: For each p-d pair, we first calculate the average ΔC value for the best N_{move} non-tabu moves in the neighborhood. Then, a temperature $temp$ is calculated such that the probability of accepting an average move is Pr . Finally, a move is selected randomly among the N_{move} moves; this move is accepted with a probability of $e^{-\Delta C/temp}$. If the current move is rejected, then another move is selected randomly from the list. If none of the moves is accepted then the best move is selected. *Insert/Split*($S, (p, d), Pr$) function (provided in Appendix A) at line 8 of Algorithm 1 returns the selected move as described.

After the candidate moves for each p-d pair are determined in this fashion between lines 7 - 15 of Algorithm 1, the move to be implemented is selected randomly out of the best N_{pair} p-d pairs with moves resulting in the highest ΔC values.

Following each insert/split move, we check if the solution can be improved by merging any pickup or delivery stops for the p-d pair involved in this move. This corrective move exploits an optimality condition for MPDPSL.

Optimality Condition. *In an optimal solution of MPDPSL where the distance matrix satisfies the triangular inequality, multiple pickup (delivery) stops of a p-d pair cannot exist on the same vehicle route without a delivery (pickup) stop of the same p-d pair in between.*

We can show that a reduction in route length can be achieved for solutions that do not satisfy the optimality condition by eliminating all but the last (first) pickup (delivery) without violating the capacity constraints. After each insert/split move, we check whether

the above condition is violated for the p-d pair under consideration and merge multiple stops into one wherever possible. In the case of multiple pickup stops of the same p-d pair, the entire load is picked up at the last stop, eliminating all other pickup stops and resulting in a reduction of route length. Conversely, all deliveries but the first one are eliminated for the case of multiple delivery stops without a pickup of the same load in between.

The insert/split phase is terminated after no improvement can be achieved over the best solution for a consecutive N_{iter1} iterations.

Algorithm 1 Insert/SplitPhase

```

1: Input:  $S_c, N_{iter1}$ 
2: Output:  $S_b$  //  $S_b$  denotes the best solution
3:  $S_b \leftarrow S_c$  //  $S_c$  denotes the current solution
4:  $T \leftarrow \emptyset$  //  $T$  denotes the list of best  $N_{pair}$  insert/split moves
5:  $i \leftarrow 0$ 
6: while  $i < N_{iter1}$  do
7:   for all  $(p, d) \in S_c$  do
8:     if  $c(Insert/Split(S_c, (p, d), Pr)) < c(maxCost(T))$  then
9:       if  $|T| = N_{pair}$  then
10:         $T \leftarrow T - maxCost(T)$  and  $T \leftarrow T \cup Insert/Split(S_c, (p, d), Pr)$ 
//  $maxCost(T)$  returns the insert/split position(s) with the largest cost from
// the list  $T$ 
11:       else
12:         $T \leftarrow T \cup Insert/Split(S_c, (p, d), Pr)$ 
13:       end if
14:     end if
15:   end for
16:   Select a pair  $(p^*, d^*)$  randomly from  $T$ 
17:   Implement selected insert/split move for  $(p^*, d^*)$ 
18:   Check Optimality Condition and update  $S_c$  if necessary
19:   if  $c(S_c) < c(S_b)$  then
20:      $S_b \leftarrow S_c$ 
21:      $i \leftarrow 0$ 
22:   else
23:      $i \leftarrow i + 1$ 
24:   end if
25: end while
26: return  $S_b$ 

```

3.3 Intra-Route Phase

On a feasible vehicle route, if a pickup node is shifted towards its corresponding delivery node, or a delivery node is shifted towards its pickup node, the precedence and capacity constraints will not be violated. Therefore, a pickup (delivery) node can be shifted towards its delivery (pickup) node, as long as the shift does not violate the route length constraint. The intra-route phase exploits this property within the blocks. For each block, the heuristic considers shifting all pickup nodes forward and all delivery nodes backward in the route, starting from the first node in the block. If a decrease in route length can be achieved, the move is implemented and the heuristic proceeds with the next block.

3.4 Block Insert Phase

In one iteration of the block insert phase, a *block* of consecutive pickup and delivery stops is removed from its current route and inserted into a new route as a whole. The capacity and precedence constraints remain intact when a block is removed from a route and inserted into another. Therefore, only route length constraints need to be checked to ensure the feasibility of solutions in this neighborhood. This phase is terminated after no improvement can be achieved over the current best solution for a consecutive N_{iter2} iterations.

The pseudocode of the block insert phase is provided in Algorithm 2. $BlockInsert(S)$ function (provided in Appendix A) at line 7 of Algorithm 2 returns the block insert move that results in the lowest ΔC value and the corresponding block insert move is performed at line 8 of Algorithm 2. In the worst case, a solution may contain nk blocks, each containing a single p-d pair. In this case, there would be nk different positions where the block can be inserted, resulting in $O(n)$ moves in the neighborhood of a particular block. Thus, for a given solution, selecting the block insert move that results in the most route length reduction is $O(n^2)$. Compared to the insert/split neighborhood, evaluating the block insert neighborhood is very fast.

3.5 Block Swap Phase

The neighborhood of this phase consists of solutions that are obtained by swapping the routes of two blocks and inserting those blocks at the best possible positions on their new routes. Similar to the block insert move, block swap moves also do not violate the

Algorithm 2 BlockInsertPhase

```
1: Input:  $S_c, N_{iter2}$  //  $S_c$  denotes the current solution
2: Output:  $S_b$  //  $S_b$  denotes the best solution
3:  $S_b \leftarrow S_c$ 
4:  $i \leftarrow 0$ 
5:  $c(S_b) \leftarrow c(S_c)$ 
6: while  $i < N_{iter2}$  do
7:    $[B^*, \ell^{B^*}] \leftarrow BlockInsert(S_c)$ 
8:    $UpdateSolution(S_c, B^*, \ell^{B^*})$ 
9:   if  $C(S_c) < C(S_b)$  then
10:     $S_b \leftarrow S_c$ 
11:     $i \leftarrow 0$ 
12:   else
13:     $i \leftarrow i + 1$ 
14:   end if
15: end while
16: return  $S_b$ 
```

precedence and capacity constraints. It suffices to check only the route length constraints to create feasible moves. In each iteration of the block swap phase, the block swap move that yields the lowest ΔC value is identified and implemented. The block swap phase is as well terminated when no improvement can be achieved over the best solution for a consecutive N_{iter3} iterations.

The pseudocode of the block swap phase is provided in Algorithm 3. *BlockSwap*(S) function (provided in Appendix A) at line 7 of Algorithm 3 returns two blocks to be swapped and the positions to be inserted in their respective routes. In a feasible solution, there are $O(n^2)$ possible block swaps, and the computational complexity of determining the best position for a block is $O(n)$. Therefore, computational complexity of selecting the pair of blocks that yield the most route length reduction is $O(n^3)$.

3.6 Tabu Embedded Simulated Annealing (TESA) Algorithm

TESA Algorithm (pseudocode provided in Algorithm 4) begins by creating an initial solution as described in Section 3.1, then it goes through several improvement phases described in Sections 3.2, 3.3, 3.4 and 3.5.

The termination of the subsequent improvement phases depends on the value of Pr ,

Algorithm 3 BlockSwapPhase

```
1: Input:  $S_c, N_{iter3}$  //  $S_c$  denotes the current solution
2: Output:  $S_b$  //  $S_b$  denotes the best solution
3:  $S_b \leftarrow S_c$ 
4:  $i \leftarrow 0$ 
5:  $c(S_b) \leftarrow c(S_c)$ 
6: while  $i < N_{iter3}$  do
7:    $[B', B'', \ell^{B'}, \ell^{B''}] \leftarrow BlockSwap(S_c)$ 
8:    $UpdateSolution(S_c, B', B'', \ell^{B'}, \ell^{B''})$ 
9:   if  $c(S_c) < c(S_b)$  then
10:     $S_b \leftarrow S_c$ 
11:     $i \leftarrow 0$ 
12:   else
13:     $i \leftarrow i + 1$ 
14:   end if
15: end while
16: return  $S_b$ 
```

Algorithm 4 TESA Algorithm

```
1: Input:  $N_{iter1}, N_{iter2}, N_{iter3}, N_{move}, N_{pair}, Pr$ 
2: Output:  $S_b$ 
3:  $S_b \leftarrow InitialSolutionHeuristic()$ 
4: while  $Pr > 0.1$  do
5:    $S_b \leftarrow Insert/SplitPhase(S_b, N_{iter1}, Pr)$ 
6:    $S_b \leftarrow IntraRouteHeuristic(S_b)$ 
7:    $S_b \leftarrow BlockInsertPhase(S_b, N_{iter2})$ 
8:    $S_b \leftarrow BlockSwapPhase(S_b, N_{iter3})$ 
9:    $Pr \leftarrow \beta Pr$ 
10: end while
11: return  $S_b$ 
```

the selection probability employed in the insert/split phase. For a given value of Pr , the algorithm executes the four phases described above subsequently before it decreases Pr by a factor of $\beta < 1$. This step is parallel to the reduction of the temperature by a cooling factor in a conventional simulated annealing algorithm. At the beginning of the algorithm, the probability of selecting inferior solutions is quite high, which allows the search to diversify and avoid the local minima. As the algorithm progresses, the probability of selecting inferior solutions decreases gradually, so that the algorithm converges to a good solution. After Pr is decreased, the algorithm starts with the insert/split phase which takes the solution from block swap phase as an input. However, if the solution is not improved in the intra-route phase, the block insert phase and the block swap phase, the insert/split phase takes the same solution from its previous execution. In order to improve diversity, if no improvement is achieved during the intra-route phase, the block insert phase and the block swap phase, the insert/split phase takes the solution obtained after tabu list size number of iterations of the block swap phase as an input.

The first version of the algorithm only consisted of a straightforward implementation of the insert/split phase with tabu search. Rather than choosing from a pool of good quality moves with simulated annealing algorithm, only the best move is chosen for a given p-d. There were three alternatives for selecting the p-d for insert/split:

Exhaustive. The p-d pair resulting with the best ΔC value is selected.

Best Deletion Saving. The p-d pair with the best deletion saving, i.e. the p-d resulting with the most cost reduction after it is removed from its route, is selected.

Random. The p-d pair for insert/split move is selected randomly.

Out of these alternatives, even though *exhaustive* approach yielded the best results, its computational complexity for one iteration is $O(n^5)$, as opposed to $O(n^4)$ for the other alternatives since the selection of the p-d has a time complexity of $O(n)$ for the *exhaustive* approach and $O(1)$ for the others.

We also tried to benchmark the performance of tabu search against a *Hill Climbing* heuristic that terminates when no improving move is found.

This experimental study indicated that tabu search is as much likely as the hill climbing heuristic to get stuck in a local optimum. This observation motivated us to employ some kind of randomization scheme.

In a conventional simulated annealing algorithm, a neighbor from all of the neighborhood solutions is selected randomly and implemented if it is accepted based on a selection

criterion. A fitness function determines the quality of the solution and returns the likelihood of that solution to be accepted. In our case, it would be extremely time consuming to evaluate all of the neighboring solutions. Therefore, we use the *list* approach where the selection criterion is implemented on the best N_{move} non-tabu results in order to ensure that the algorithm does not spend time in parts of the solution space that are not rewarding. For small values of N_{move} , the algorithm runs with the risk of revisiting the same solutions; utilizing a tabu structure ensures that the algorithm does not revisit the same solution. This overcomes one of the major weaknesses of simulated annealing, that it lacks any memory features to track previously visited solutions and the solution quality improves considerably after the introduction of the probabilistic selection criterion. The resulting algorithm combines powerful features of both tabu search and simulated annealing. The computational results reported in the next section demonstrate that this algorithm is capable of producing good solutions for MPDPSL in reasonable time.

3.7 Computational Results

3.7.1 Parameter Tuning for TESA Algorithm

Our preliminary experiments indicate that five of the parameters used in TESA algorithm have a significant impact on both solution quality and CPU time: N_{iter1} , N_{iter2} , N_{iter3} , N_{pair} and N_{move} . For the purpose of parameter tuning, we only consider these parameters. To simplify the experimental design, we use, in a given setting, the same values for the maximum number of nonimproving iterations in any phase, i.e. N_{iter1} for the Insert/Split phase and N_{iter2} for Block Insert phase and N_{iter3} for Block Swap phase; we denote this parameter as N_{iter} . N_{move} determines the size of the list that keeps the least costly moves for a selected p-d pair; the likelihood of selecting a poor quality move increases as N_{move} gets larger. N_{pair} determines the size of the list that keeps the best pairs for the current solution in implementing a move (lines 9 - 10 of Algorithm 1). Similarly, when N_{pair} gets larger, the likelihood of selecting an inferior pair is increased.

The computational study for these parameters is conducted with the multi-vehicle version of the randomly generated problems in Nowak et al. [10]. Parameter tuning is done for the set of problems of 100 p-d requests with a load range of 0.51-0.6 truckload. For $N_{iter} \in \{25, 50, 75, 100, 125\}$, $N_{move} \in \{5, 7, 10, 15\}$, and $N_{pair} \in \{1, 3, 5, 7, 10\}$, we conduct the tuning study for 100 possible parameter settings. For a given setting, the algorithm is restarted 20 times. Based on preliminary tests, the value of β is set to $1/3$

throughout the experiment. To evaluate the quality of a parameter setting, we use the average improvement attained over the route length of the initial solution with no splits (see Section 3.1) as the effectiveness measure and the average CPU time as the efficiency measure. Computational experiments are conducted on a single core of a computer with Intel Core2Quad Q8200 @2.33 GHz CPU and 3.46GB of RAM.

According to the results of our study:

- The most influential parameter is the maximum number of nonimproving iterations, N_{iter} , as the improvement in the route length increases by 2% when N_{iter} is increased from 25 to 125. On the other hand, The solution quality changes by only 0.2% between the best and the worst settings of N_{pair} and N_{move} . The CPU time also increases when the value of each of these three parameters is increased.
- When the effect of each parameter is inspected individually (see Figures 3.1, 3.2, 3.3), $N_{iter} = 125$, $N_{move} = 10$ and $N_{pair} = 5$ appears to be the best setting of parameters.
- Inspecting the combined effect of N_{iter} and N_{pair} (see Figure 3.5), and the combined effect of N_{iter} and N_{move} (see Figure 3.4), it is clear that the effect of N_{iter} overweighs and best results are attained when N_{iter} is as large as possible (i.e. $N_{iter} = 125$). According to the combined effect of N_{move} and N_{pair} in Figure 3.6, $N_{move} = 15$ and $N_{pair} = 3$ yields the best solution quality. Thus, when the combined effect is considered, the best setting for all three parameters appears to be $N_{iter} = 125$, $N_{move} = 15$ and $N_{pair} = 3$.

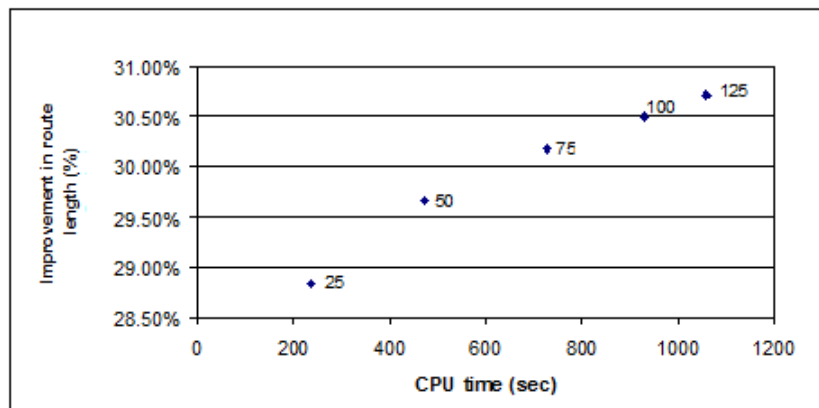


Figure 3.1: Improvement in route length versus CPU time for different values of N_{iter} .

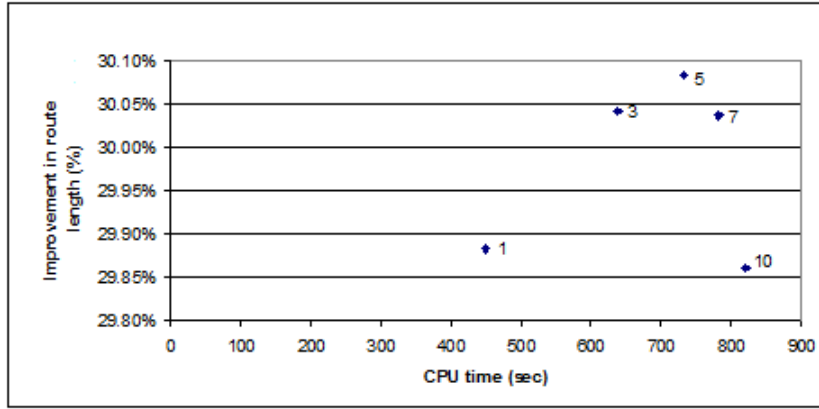


Figure 3.2: Improvement in route length versus CPU time for different values of N_{pair} .

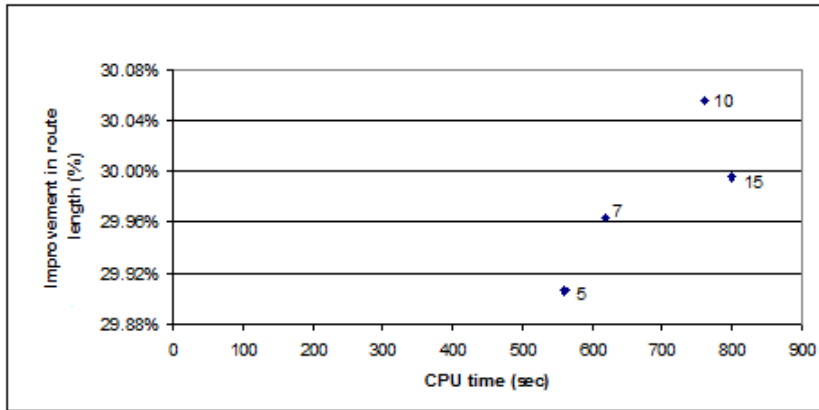


Figure 3.3: Improvement in route length versus CPU time for different values of N_{move} .

Taking into account the trade-off between the improvement in the route length and CPU time, it would be more reasonable to set the value of N_{iter} based on the availability of computing power. In essence, one might prefer a smaller value of N_{iter} if the computing resources are restricted and CPU time is an important concern. The values of N_{pair} and N_{move} are dependent on the problem size, i.e. the number of p-d requests. Therefore, we adjust the values of these parameters in proportion to the number of p-d requests in a given instance.

Another important parameter that affects the CPU time is the number of restarts. We do not include this parameter directly in the parameter tuning study. Instead, we use each restart as a sampling procedure since each restart of the algorithm randomizes the selection of moves. A high number of restarts increases the likelihood of diversification

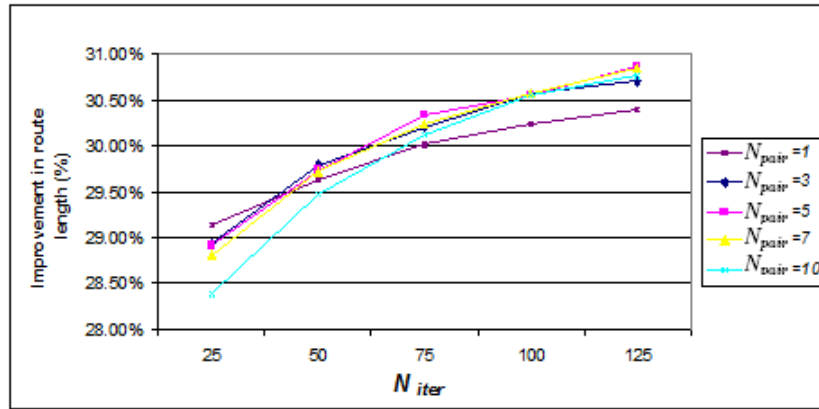


Figure 3.4: Combined effect of N_{iter} and N_{pair} on the improvement in route length.

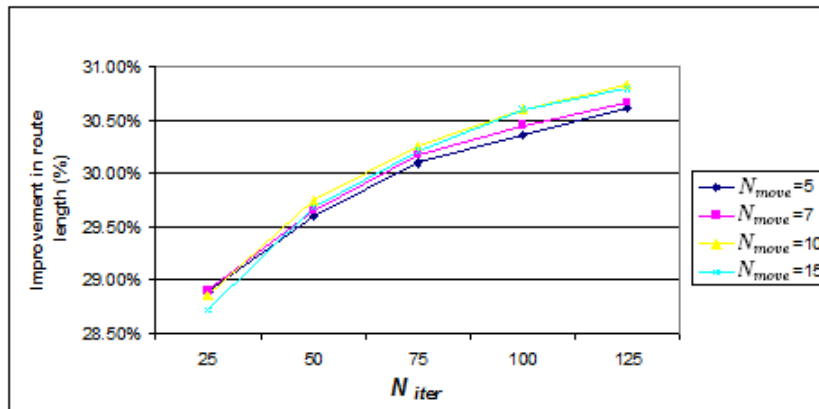


Figure 3.5: Combined effect of N_{iter} and N_{move} on the improvement in route length.

among the obtained solutions. In order to validate this anticipation, we closely investigate the improvement pattern through the restarts of the algorithm over a maximum of 100 restarts. In Figure 3.7, we show the route length obtained at every restart of the algorithm along with the best route length obtained until after that restart. In these three examples, we observe that the best route length is obtained at restart 23 (in part (a)), restart 47 (in part (b)) and restart 96 (in part (c)). This observation validates the significance of restarts and the effect of randomization on the solution quality through the diversification mechanism induced by restarts. It is clear that the solution quality might improve when more effort is spent by increasing the number of restarts. As in the case of N_{iter} , one should consider the trade-off between the CPU time and number of restarts with respect to the availability of computing resources.

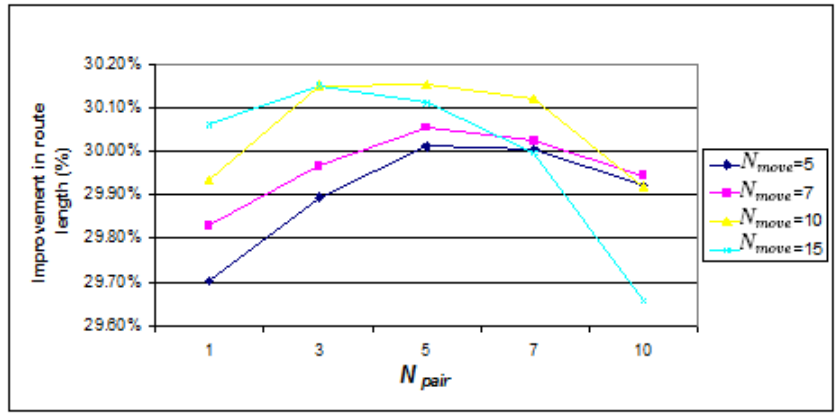
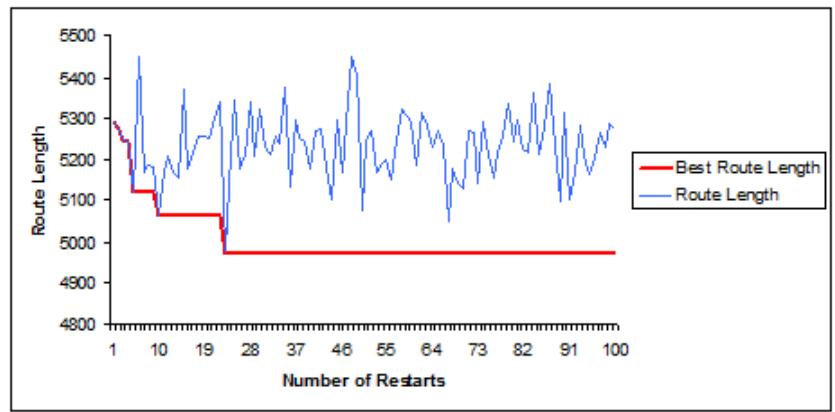
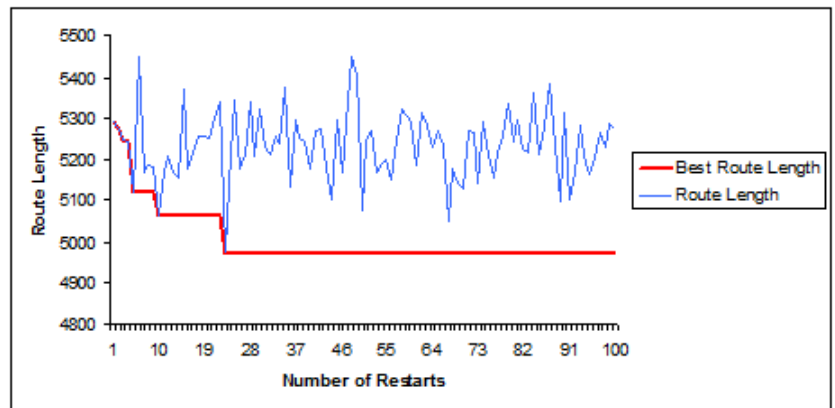


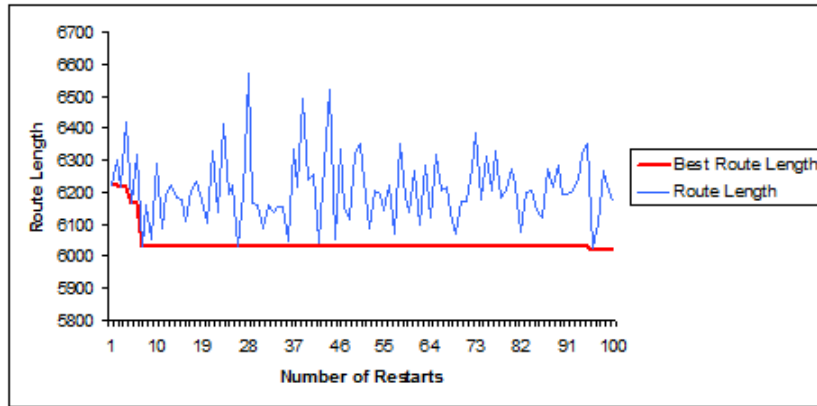
Figure 3.6: Combined effect of N_{move} and N_{pair} on the improvement in route length.



(a)



(b)



(c)

Figure 3.7: The pattern of route length improvement through the restarts of the algorithm.

3.7.2 Computational Experiments on Test Problems

Nowak et al. [10] have investigated the improvement in route length when splits are allowed over the route length of the solution with no splits. The aim of their analysis is to show that one may benefit from splitting some p-d requests (i.e. stopping at the origin and destination of the requests more than once) instead of visiting the p-d pair only once as in the traditional setting of PDP. In this study, we extend Nowak et al.'s analysis to the multiple vehicle version of the problem. Moreover, we analyze the benefit of splitting loads on other PDP instances from the literature. As detailed below, Nowak et al.'s instances have special characteristics that do not exist in other PDP settings. In order to explore the impact of problem characteristics on the benefit of splitting loads, we use another set of problem instances adopted from Ropke and Pisinger [13] whose characteristics are significantly different. We also verify the quality of our algorithm through a comparative analysis against the PDP results given in Nowak et al. [10]. Since these results are for the single vehicle case, we convert our solutions to a single route using a very simple algorithm.

Below, we describe the two problem sets used in our analysis and how they are adapted to create instances for MPDPSL:

1. The first set consists of the randomly generated problems in Nowak et al. [10]. In generating this set of problems, the authors have used the following idea: they randomly generate a set of pickup points and another set of delivery points; then, they create a set of p-d requests from every pickup point to every delivery point.

As a result, a p-d request is still one-to-one but there are multiple deliveries from a pickup location as well to a delivery location. In addition, Nowak et al. considers a single-vehicle problem which constructs a single route as a solution. Since we study the multi-vehicle case, our solution consists of multiple routes, one for each vehicle. This setting requires introducing a maximum route length restriction for each vehicle route. For each problem instance from Nowak et al., we introduce a maximum route length as a function of the distances between points.

2. In a traditional PDP setting, each physical pickup point is usually associated with a single delivery point. In this respect, the problems in Nowak et al. may not be considered as representative of a traditional PDP setting in terms of the spatial distribution of the pickup and delivery points. For problems that are more representative of a traditional PDP setting, we resort to Ropke and Pisinger [13]. In this set of problems, all p-d requests are generated one-to-one between a randomly generated pickup point and a randomly generated delivery point. Ropke and Pisinger suppose that each vehicle starts the route from a designated point considered to be the vehicle's depot. However, in MPDPSL we suppose that there exists only one depot from where all vehicles start their routes. Therefore, based on the network data in Ropke and Pisinger's instances, we have created a single vehicle depot at the center of the two-dimensional space on which the points are generated randomly. The maximum route length is used as specified in this study.

To the best of our knowledge, for both sets of problems, the results presented in this section are the first results for MPDPSL in the literature.

3.7.3 Analysis of the Benefit Obtained by Split Loads

In Nowak et al. [10], the benefit obtained by splitting loads is studied for different load size intervals. Their results clearly show that the improvement in the route length is closely related to the size of the loads. When the loads are within 0.51-0.60 of vehicle capacity, the improvement in the route length is more significant compared to other ranges of the load size. We want to understand if this finding is still true when the problem is solved for multiple vehicles instead of a single vehicle. For this analysis, we solve each problem instance using two versions of our algorithm: the original version that allows split loads and a modified version which does not perform any split moves.

In Table 3.1, we present the percentage improvement in the route length for the algorithm with splits over that of the no-split version for the three problem sets in Nowak et

Number of Requests	Location Set	Average Imp. (%)	Average
75	1	33,11	32,04
	2	30,81	
	3	32,21	
100	1	33,09	32,45
	2	32,01	
	3	32,26	
125	1	32,28	32,07
	2	31,54	
	3	32,38	

Table 3.1: Improvement in the route length obtained by split loads for problems in Nowak et al. [10] for a load size range of [0.51-0.6] truckload.

al. [10]. The improvement in the route length is more than 30% on average when the load size range is 0.51-0.60. The observed level of improvement is in line with Nowak et al.’s findings where the improvements are around 25% for the same load size range.

In order to study the benefit of splitting loads in a more traditional PDP setting where the pickup and delivery points are distributed over the service area, we solve the problems in Ropke and Pisinger [13] to obtain both the split and no-split route lengths. To observe the difference in route length improvement between different load size ranges, we have studied the load range 0.25-1.00 (using the loads in the original problem data) and the load range 0.51-0.60 (for which we have generated new load data randomly). Complete results are given in Tables B.1-B.4 of Appendix B. For the problems with 50, 100 and 250 requests, we attain the best results over 20 restarts while the problems with 500 requests is solved only with five restarts and with $N_{iter} = 10$ due to excessive CPU time for a single restart of the algorithm. The resulting CPU times are provided in Tables B.5 - B.6 of Appendix C. A summary of results grouped by the number of requests and load size of problem instances is presented in Figure 3.8. In this figure, we observe that

- the improvement in route length for the load range 0.51-0.60 is more significant when compared to the load range 0.25-1.00, and
- the improvement in route length increases as the problem size increases.

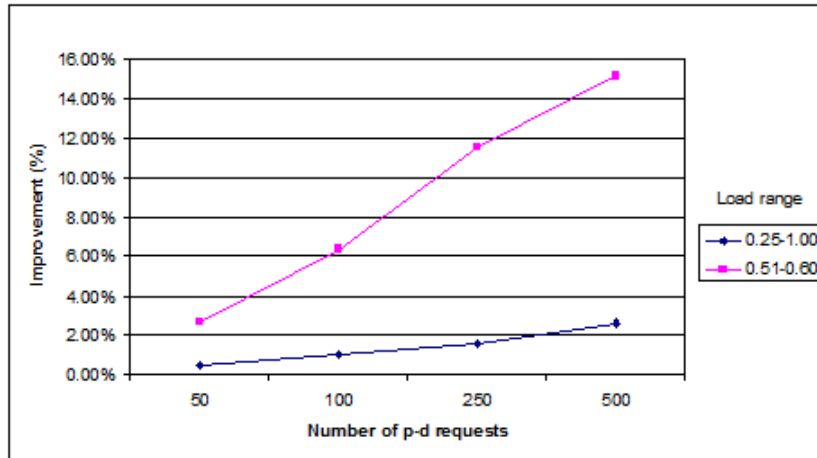


Figure 3.8: Effect of the load range on the improvement in route length.

A comparison among the two load ranges help us conclude that the improvement in route length due to split loads is more significant when the load range is just above one half of a truck load (0.51-0.60) compared to a much wider load range. This observation is in parallel with the empirical analysis in Nowak et al.[10]. Yet, it is also clear that this difference becomes more significant as the problem size increases and the magnitude of improvement is not as much as it is observed in Nowak et al. Therefore, in addition to the load range size, we observe that

- spatial distribution of the pickup and delivery requests and
- number of p-d requests in the problem

also play an important role on the level of improvement in route length. In Nowak et al., although the problem is a one-to-one PDP, each pickup (delivery) point is associated with several requests. In the problems by Ropke and Pisinger [13], each pickup (delivery) point is associated with only one delivery (pickup) point. We also note that the observed improvement is not due to the algorithm used to solve the problems as the level of improvement in Nowak et al.'s experimental results is around 25-30% while it is in the range of 30-33% in ours. In addition, we suspect that the multi-vehicle version of the problem may be even more prone to improvements obtained by split loads.

The importance of spatial distribution becomes even more apparent when we look into the results from the Ropke and Pisinger [13] instances in more detail. These instances are designed in three categories according to the spatial distribution of the pickup and delivery

locations: uniform, semi-clustered and clustered. In Figure 3.9, we display the improvement in route length both by load size and problem category. While the importance of load size is clear in this figure, one can observe that the benefit of split loads is also affected significantly by the problem category. The largest improvement in route length is realized for clustered data, especially for larger problem sizes. Although the difference between semi-clustered and uniform data is not very significant for small problems, as the number of p-d pairs in the problem increase, semi-clustered problems also benefit greatly from splitting loads.

Overall, the results indicate that the benefit to be gained from load splitting is more pronounced for problems with clustered data, even when the clustering is partial. In clustered problems, vehicle routes also tend to appear in clusters, which increases the likelihood of having multiple vehicle routes in close vicinity which facilitates load splitting. Similarly, the number of splitting options are higher for larger problem sizes. This observation may help explain the results obtained for problem instances from different categories and problem sizes.

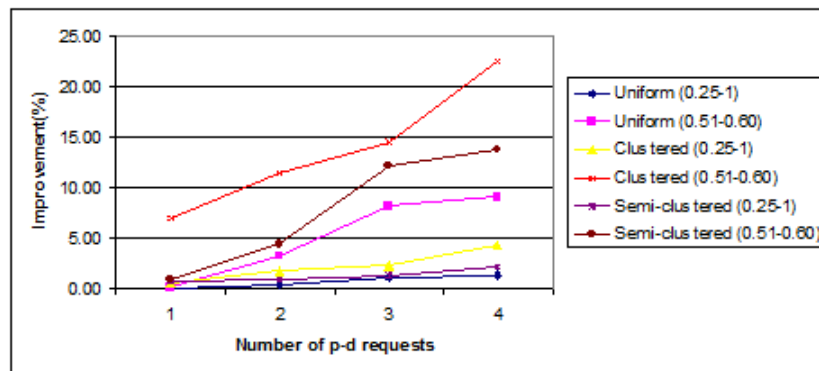


Figure 3.9: Effect of the load range and spatial distribution of points on the improvement in route length.

We also note that this is the first study to report results on the problems in Ropke and Pisinger [13] for the case of split loads.

3.7.4 Comparative Analysis of Algorithm Performance

We evaluate the performance of our algorithm on the single vehicle version of the problem using a comparative analysis against the results presented in Nowak et al. [10]. The aim of our analysis is two-fold; we want to

- validate the quality/strength of our algorithm in order to justify the use of our algorithm in obtaining the results in Section 3.7.3, and
- show that our algorithm has the potential to be used for the single vehicle case by demonstrating an improvement upon the results obtained in Nowak et al. [10].

We particularly note that our algorithm is designed for the multiple vehicle version, and we do not make a special effort to tailor it to solve the single vehicle case. In order to obtain a single vehicle route from the multi-vehicle solution of the algorithm, we use a very simplistic greedy approach. Among all vehicle routes in the solution, we identify the two that yield the maximum route length decrease when merged into a single route. After merging these two routes, at every iteration we add a new route to the combined route such that the incremental route length increase is minimized. The procedure stops when all routes are merged into a single vehicle route.

For the three sets of problem instances (i.e. 75,100,125-request sets), we solve the single vehicle problems with the load factor 0.51-0.60; the results are presented in Tables 3.2-3.4. To make a fair comparison, we account for the difference in the computing power of our computer with the one used in Nowak et al. [10]. For this purpose, we present our results with two different levels of computational effort. In the scaled time approach, for each problem set, we scale the average CPU time reported in Nowak et al. by the difference in the computing power of the computers using the method presented in Dongarra [5]. As our computer is two times more efficient than the one in Nowak et al., we use half of the average CPU time reported in Nowak et al. as the scaled time. In the equal time approach, we use as much CPU time as reported in Nowak et al. without any scaling. In Tables 3.2-3.4, the first three columns describe the problem characteristics (Location Set and Load Size Set) based on Nowak et al. and the length of the route obtained with their algorithm. The fourth and fifth columns respectively show the length of the route and percentage improvement over Nowak et al.'s solution using the scaled time approach. The sixth and seventh columns show the same results when the CPU time is not scaled.

According to computational experiments reported in Tables 3.2-3.4, we obtain the following results:

- For the problem set with 75 requests,
 - in scaled time (12.75 minutes), we have improved the route length of 9 problems (out of 15) and the average improvement is 2.10% while we are only

Problem (Nowak et al.)			Scaled Time		Equal Time	
Location	Load Size	Route Length	Route Length	Imp. (%)	Route Length	Imp. (%)
1	1	3830.123	3840.602	-0.27	3840.602	-0.27
1	2	3857.117	3859.936	-0.07	3828.613	0.74
1	3	3810.502	3831.256	-0.54	3809.375	0.03
1	4	3799.324	3833.559	-0.90	3833.559	-0.90
1	5	3868.957	3887.298	-0.47	3855.453	0.35
2	1	3313.483	3171.766	4.28	3171.766	4.28
2	2	3296.364	3162.015	4.08	3144.403	4.61
2	3	3203.245	3210.825	-0.24	3199.044	0.13
2	4	3266.417	3180.605	2.63	3180.605	2.63
2	5	3332.589	3188.812	4.31	3146.602	5.58
3	1	4058.369	3990.376	1.68	3954.308	2.56
3	2	4172.418	3926.42	5.90	3926.42	5.90
3	3	4090.647	3934.566	3.82	3934.566	3.82
3	4	4110.389	3936.68	4.23	3936.68	4.23
3	5	4052.233	3925.661	3.12	3908.682	3.54
			9	2.10	13	2.48

Table 3.2: Results for the single vehicle problems with 75 requests from Nowak et al. [10].

0.90% away from the best known solution in the worst case;

- in equal time (25.50 minutes), we have improved the route length of 13 problems and the average improvement is 2.48 %.
- For the problem set with 100 requests,
 - in scaled time (25.60 minutes), we have improved the route length of 13 problems (out of 15) and the average improvement is 3.30% while we are only 0.82% away from the best known solution in the worst case;
 - in equal time (56.20 minutes), we have improved the route length of 14 problems and the average improvement is 3.60 %.
- For the problem set with 125 requests,
 - in scaled time (47.95 minutes), we have improved the route length of 13 problems (out of 15) and the average improvement is 4.45% while we are 1.37% away from the best known solution in the worst case;
 - in equal time (95.90 minutes), we have improved the route length of 14 problems and the average improvement is 4.71%.

These results indicate that we have improved most of the route lengths reported in Nowak et al. [10] even with an algorithm which is not tailored for the single vehicle version. Therefore, our algorithm can be considered as an efficient and effective algorithm for solving not only the multiple vehicle but also the single vehicle problems as well.

Problem (Nowak et al.)			Scaled Time		Equal Time	
Location	Load Size	Route Length	Route Length	Imp. (%)	Route Length	Imp. (%)
1	1	5073.395	5016.763	1.12	5014.602	1.16
1	2	5036.547	5077.608	-0.82	5077.608	-0.82
1	3	5029.381	5033.059	-0.07	5024.470	0.10
1	4	5012.974	4965.867	0.94	4965.867	0.94
1	5	5130.154	5022.422	2.10	5022.422	2.10
2	1	4450.058	4249.250	4.51	4216.749	5.24
2	2	4484.466	4302.366	4.06	4302.366	4.06
2	3	4473.387	4292.695	4.04	4268.062	4.59
2	4	4424.569	4283.759	3.18	4259.868	3.72
2	5	4559.259	4272.694	6.29	4247.892	6.83
3	1	5294.367	5029.347	5.01	5029.347	5.01
3	2	5371.740	5158.324	3.97	5127.881	4.54
3	3	5216.797	5149.056	1.30	5115.153	1.95
3	4	5467.788	5132.782	6.13	5132.782	6.13
3	5	5572.472	5141.317	7.74	5097.744	8.52
			13	3.30	14	3.60

Table 3.3: Results for the single vehicle problems with 100 requests from Nowak et al. [10].

Problem (Nowak et al.)			Scaled Time		Equal Time	
Location	Load Size	Route Length	Route Length	Imp. (%)	Route Length	Imp. (%)
1	1	6020.046	5924.608	1.59	5924.608	1.59
1	2	5938.943	6020.369	-1.37	6008.256	-1.17
1	3	5977.69	5929.926	0.80	5929.926	0.80
1	4	6138.936	6022.086	1.90	6017.348	1.98
1	5	6024.26	6028.898	-0.08	5996.25	0.46
2	1	5717.536	5452.732	4.63	5410.327	5.37
2	2	5745.378	5494.878	4.36	5470.25	4.79
2	3	5667.263	5456.303	3.72	5456.303	3.72
2	4	5778.58	5428.613	6.06	5409.127	6.39
2	5	5780.014	5471.003	5.35	5430.16	6.05
3	1	6934.046	6322.772	8.82	6272.689	9.54
3	2	6918.162	6318.539	8.67	6318.539	8.67
3	3	6607.296	6330.95	4.18	6330.95	4.18
3	4	7239.787	6412.622	11.43	6404.65	11.54
3	5	6776.373	6320.865	6.72	6320.865	6.72
			13	4.45	14	4.71

Table 3.4: Results for the single vehicle problems with 125 requests from Nowak et al. [10].

Chapter 4

MPDPSL with Outsourcing Option

In most business practices, seasonality and volatility of demand play a great role on the logistics decisions of a company. Instead of having a fleet on standby large enough for the busiest time of the month or the year, most logistics and courier companies use the outsourcing option for some of their shipments. Therefore, they are able to manage the same amount of demand with a much smaller fleet. On the other hand, even when the company has adequate resources, sometimes, it is simply more cost efficient to outsource because of the network conditions or the nature of demand.

In the MPDPSL with Outsourcing Option (MDPDPSL-O), we have, for any p-d pair, the option of outsourcing for a fixed charge. If a p-d pair is chosen for outsourcing then it would not be visited in any of the routes in the solution and the fixed charge for outsourcing is added to the objective function value. In addition to the model presented in Chapter 2, the modified mathematical model becomes,

$$\text{Minimize } \sum_{k \in K} \sum_{i \in V} \sum_{j \in V} d_{ij} x_{ij}^k + \sum_{i \in P} f_i z_i \quad (4.1)$$

$$\text{subject to } \sum_{k \in K} y_i^k + z_i = 1 \quad \forall i \in P \cup D \quad (4.2)$$

$$z_i = z_{i+n} \quad \forall i \in P \quad (4.3)$$

$$z_i \in \{0, 1\} \quad \forall i \in P \cup D \quad (4.4)$$

where $z_i, \forall i \in P \cup D$, denotes a binary decision variable; $z_i = 1$ if demand of i is outsourced; 0, otherwise. $f_i, \forall i \in P$, denotes the fixed charge of outsourcing the demand of i . The modified objective function (4.1) has an additional term, $\sum_{i \in P} f_i z_i$ that adds the

outsourcing cost to the transportation cost. Constraint (4.2) is also modified to ensure that the demand is either satisfied by the company *or* outsourced while constraint (4.3) ensures that when a pickup stop is outsourced, its corresponding delivery stop is outsourced as well. Constraint (4.4) is the domain constraint for $z_i, \forall i \in P \cup D$. The rest of the constraints from the model presented in Chapter 2 are also in effect.

4.1 Modified TESA Algorithm for MPDPSL-O

We already have an efficient and effective algorithm for MPDPSL, and we can modify the algorithm for this problem by a few simple tweaks. The decision of outsourcing a p-d pair is not much different than the decision of which route/s to insert/split the pair. We modify the insert/split neighborhood in order to consider the outsourcing option, when we are searching for a position to insert/split a p-d pair. If the outsourcing option is cost effective and can compete with other insert/split positions, then it is inserted to the list from which we choose the *move* to be implemented, thus having a chance of selection.

In the TESA Algorithm (see Algorithm 4), recall that Pr denotes the probability of an average quality move to be accepted. In this respect, during the early stages of the algorithm, even if the outsourcing option is cost efficient, it might not be chosen. However, once Pr gets smaller as the algorithm progresses, if the outsourcing option is cost efficient as opposed to the other moves, it will have a better chance of being selected. On the other hand, once we choose to outsource a p-d pair, the solution is not stuck with that decision throughout the algorithm. As in the original algorithm where each p-d pair is considered for insert/split positions, we consider each p-d pair in the modified algorithm as well. Even after we chose to outsource a p-d pair, we can still search for insert/split positions. Therefore, if we can find a suitable position, any outsourced pair can go back to the solution.

The algorithm usually chooses to outsource pairs with advantageous fixed charges. However, because of the randomness in the algorithm, it is not guaranteed that such pairs will be outsourced. As an addition to the original algorithm, we incorporate a correction phase to further improve the solution. After the final solution is reached, we check if outsourcing those pairs would result in an improvement in the objective function value for each pair in the final solution. If we can improve the objective function value, we remove those pairs from the routes in the solution and utilize the outsourcing option.

4.2 Computational Results

For most companies, in-house cost for carrying the demand is readily available but outsourcing cost might not always be at hand. However, since the outside carrier will travel the distance from the pick-up point to the delivery point in order to carry the demand, it would not be inaccurate to assume that the fixed charge for that p-d pair will be somewhat proportional to the distance between the pickup and delivery points. Considering that the vehicles of the outside carrier might have to leave and return to a depot location, we generate fixed charges for the problem instances from Ropke and Pisinger [13], using the following schemes.

Direct Distance. We suppose that it is not important for the company from where the vehicle of the outside carrier reaches the pickup point and to where it retreats after delivering the demand to the delivery point. Therefore, we only take the distance between the pickup and delivery points, and the fixed charge for outsourcing becomes a function of d_{pd} as,

$$f_i = \alpha d_{p_i d_i}, \quad \forall i \in P.$$

Triangular Distance. We suppose that the vehicle of the outside carrier leaves from a depot location and retreats to the depot after picking the demand from the pickup point and delivering it to the delivery point. Therefore, we calculate the distance from the depot to the pickup point plus the distance from the pickup point to the delivery point and finally the distance from the delivery point to the depot, and the fixed charge for outsourcing becomes a function of d_{pd} as,

$$f_i = \gamma(d_{0p_i} + d_{p_i d_i} + d_{d_i 2n+1}), \quad \forall i \in P.$$

Note that d_{ij} denotes the distance between i and j and α and γ denote constant scalars.

The costs generated according to these schemes may not match the cost scheme of an outside carrier; however, we will observe the whether behavior of the algorithm changes under these different cost schemes. Also, by using alternative values for scalars α and γ , we will create scenarios when the outsourcing cost is lower and higher relative to the company's in-house transportation cost.

4.2.1 Parametric Settings for MPDPSL-O

We treat the outsourcing option as any other move in the original algorithm. Therefore, the basic principles and dynamics of the original algorithm remains intact. We do not tamper with the original parameters of the algorithm discussed in Section 3.7.1. The computational performance is not our main concern at this point, and we care about the solution structure and effectiveness.

When a company has outsourcing options, it faces tactical decisions concerning the fleet size and a cost-effective pricing scheme for the outsourcing option. In order to analyze the effect of the outsourcing cost on the solutions, we prepare scenarios based on different outsourcing cost schemes. We assume that the results of the original algorithm without the outsourcing option is the in-house cost of the company for satisfying all the requests using its own resources. For the problem instance p , let $c(S_p)$ be the objective function value obtained by the original TESA Algorithm for the solution S_p . Then for instance p , α and γ values are calculated as follows:

$$\alpha = \frac{\omega c(S_p)}{\sum_{j=1}^n d_{p_j d_j}},$$

$$\gamma = \frac{\omega c(S_p)}{\sum_{j=1}^n (d_{0p_j} + d_{p_j d_j} + d_{d_j 2n+1})}.$$

where ω denotes a constant scalar for $c(S_p)$.

The outsourcing cost plays a crucial role on the solution structure and the number of p-d pairs outsourced. In order to analyze the effect of lower and higher outsourcing costs as opposed to the in-house cost, we use three different cost schemes where the sum of outsourcing fixed charges ($\sum_{i \in P} f_i$) is

- less than $c(S_p)$,
- equal to $c(S_p)$ and
- more than $c(S_p)$.

We set $\omega \in \{0.75, 1.0, 1.25\}$ so that the sum of fixed charges would fit these cost schemes. If we choose to outsource all pairs, we would have to pay $\sum_{i \in P} f_i = \omega c(S_p)$. For example, when $\omega = 0.75$,

$$\sum_{i \in P} f_i = 0.75c(S_p)$$

in the case of the first cost scheme.

4.2.2 Benchmarking MPDPSL-O Solutions with MPDPSL Solutions

We conduct our experiments for MPDPSL-O on Ropke and Pisinger [13] instances. We test our algorithm on the problems with load ranges 0.25-1 and 0.51-0.60 for both *direct* and *triangular* distance schemes. The aggregated results from 50, 100, 250 and 500 pair instances are presented in Tables 4.1 - 4.4 where $Outs.(\%)$ denotes the percentage of the pairs outsourced and $Imp.(\%)$ denotes the improvement obtained with MPDPSL-O solution with respect to MPDPSL solution.

ω	0.75		1		1.25	
	Outs. (%)	Imp. (%)	Outs. (%)	Imp. (%)	Outs. (%)	Imp. (%)
50	83.17	25.62	29.50	10.89	13.50	5.36
100	78.08	26.07	31.25	10.85	12.42	4.99
250	83.43	26.16	32.67	9.98	10.73	3.75
500	89.07	25.81	38.55	7.93	10.82	2.41
Average	83.44	25.91	32.99	9.91	11.87	4.13

Table 4.1: Results for 0.25-1 load range with *triangular* distance scheme from Ropke and Pisinger. [13].

ω	0.75		1		1.25	
	Outs. (%)	Imp. (%)	Outs. (%)	Imp. (%)	Outs. (%)	Imp. (%)
50	93.17	25.24	37.50	9.36	17.33	4.01
100	92.17	24.97	38.08	8.04	15.50	2.61
250	94.17	24.58	34.90	6.10	11.67	1.70
500	98.97	24.82	34.50	3.41	7.67	0.41
Average	94.62	24.90	36.25	6.73	13.04	2.18

Table 4.2: Results for 0.25-1 load range with *direct* distance scheme from Ropke and Pisinger. [13].

ω	0.75		1		1.25	
	Outs. (%)	Imp. (%)	Outs. (%)	Imp. (%)	Outs. (%)	Imp. (%)
50	87.50	25.09	27.67	9.46	9.83	4.21
100	79.08	25.02	23.92	8.89	8.17	4.01
250	84.50	24.55	21.10	6.81	7.10	2.71
500	85.47	24.68	30.23	5.71	7.37	1.46
Average	84.14	24.84	25.73	7.72	8.12	3.10

Table 4.3: Results for 0.51-0.60 load range with *triangular* distance scheme from Ropke and Pisinger. [13].

Let us now look at the impact of different cost schemes on the total cost of the solutions:

- Recall that for $\omega = 0.75$, the cost of outsourcing all pairs in any instance p would be around $0.75c(S_p)$. Therefore, by outsourcing all pairs when $\omega = 0.75$, the algorithm

ω	0.75		1		1.25	
	Outs. (%)	Imp. (%)	Outs. (%)	Imp. (%)	Outs. (%)	Imp. (%)
50	91.00	24.52	33.67	8.56	14.17	2.72
100	89.42	24.09	30.33	6.52	13.00	2.56
250	87.10	22.68	26.23	4.76	9.60	1.37
500	97.23	23.83	24.50	1.67	6.63	-0.25
Average	91.19	23.78	28.68	5.38	10.85	1.60

Table 4.4: Results for 0.51-0.60 load range with *direct* distance scheme from Ropke and Pisinger. [13].

could improve by 25%. However, because of the principles of the algorithm, it tries to improve by inserting and splitting. That is the reason why, though very close, the algorithm usually cannot improve by 25% on average. On the other hand, in some cases, we can observe that the algorithm combines its strengths and the advantageous fixed charges to improve beyond 25% on average.

- When $\omega = 1.0$, outsourcing all pairs for any instance p would result in an objective function value around $c(S_p)$. However, it is clear that by making use of the outsourcing option for 25 – 33% of all pairs, the algorithm improves MPDPSL results by 5 – 10%.
- When $\omega = 1.25$, outsourcing option is very expensive since outsourcing all pairs would result in an objective function value around $1.25c(S_p)$ for any instance p . In that case, one would expect the objective function values for MPDPSL-O and MPDPSL would be more or less the same since almost none of the pairs would be outsourced due to the expensive outsourcing option. However, the algorithm is still able to improve by 1 – 4% since it utilizes the outsourcing option on 10 – 13% of all pairs.

It is evident that the algorithm on average improves more with the *triangular* distance scheme than the *direct* distance scheme for all ω values even though the *triangular* distance scheme outsources less than the *direct* distance scheme. We believe that the main reason for this outcome lies in the structure of the cost scheme: the pricing scheme of the outsourcing option with *triangular* distances and the pricing scheme of the algorithm are not quite alike. This creates a discrepancy between the fixed charges and the transportation cost of the algorithm for most pairs. We may claim that; the outside carrier underestimates or overestimates the fixed charges, and the algorithm takes advantage of this fact. On the other hand, with the *direct* distance scheme, the pricing of the outsourcing option is more similar to the pricing scheme of the algorithm, therefore it is more challenging for

the algorithm to take advantage of the discrepancy between the transportation cost and the outsourcing cost. As a result, the algorithm improves more with the *triangular* distance scheme even though it outsources less number of p-d pairs.

The load ranges and number of p-d pairs also play an important role on the level of improvement attained with outsourcing. The algorithm improves more in the 0.25-1 load range than in the 0.51-0.60 load range. Also, as the number of pairs get larger, the improvement diminishes. There are two possible reasons of these outcomes:

- the original algorithm is more attuned to splitting in the 0.51-0.60 load range, and
- the original algorithm performs better with more pairs.

As we compare the results of the modified algorithm MPDPSL-O with those of the original algorithm for MPDPSL, it is clear that the modified algorithm improves better in the 0.25-1.0 load range and with less number of pairs where there is more room to improve.

Chapter 5

MPDPSL with Cycles without Depot

Through our communications with an international maritime transportation agency, we have realized that part of their transportation network problem can be modeled as a variant of MPDPSL that has cyclic routes but no depot location. Hence, we call this problem MPDPSL with Cyclic Routes without Depot (MPDPSL-C). In their transportation network, most vessels work on a cyclic route of ports throughout a certain period of time. For example, a vessel visits port A , moves to port B and so on and so forth, following a series of several port visits, it then comes back to port A after a month and may repeat this cycle 12 times in a year. At the beginning of each term (season, year, etc.), new cycles (routes) are published and these cycles are determined by the demand information from the previous terms. Therefore, we can perceive those cycles as tactical decisions to be made.

Creating cycles is the essence of the problem though not the extent of it. With given demand information for a set of pickup and delivery pairs and known vessel capacities, the algorithm could be employed to generate cyclic routes. However, generating cycles only deals with one part of the problem; in order to solve the whole problem, we realize that we have to address the following issues as well:

- The amount of demand between different pairs of ports varies significantly. For instance, the demand from port A to port B might be 1 container while the demand from port C to port D might be as much as 4000 containers. In our case-study, including all the demand in the input would result in a MPDPSL-C instance having nearly 560 pairs; however, excluding the demand with less than 50 containers results in an instance with 208 pairs. In our implementation, we choose to eliminate smaller amount of demand. Therefore, the range of the amount of the demand in-

cluded in the MPDPSL-C input turns out to be an important decision as it affects the number of pairs in the problem which in turn changes the computational effort and implementability of the solution.

- In MPDPSL-C, we may assume that all vessels have the same capacity. In real life however, there are bunch of vessels available with various capacity. Obtaining a MPDPSL-C solution with our algorithm, we also have to make sure that the demand figures can be satisfied by the available vessels.
- Even though there is a limit to the maximum cycle length, MPDPSL-C solution does not always contain cycles that are close to the limit. The demand information is given for a fixed period of time. For instance, if a vessel has a cycle length of two months with the demands calculated for one month, the cycle must be adjusted so that the demands fit to the cycle length.

In order to solve the problem, we need to implement a larger framework in which we iteratively solve the MPDPSL-C problem by updating the obtained cycles to integrate the eliminated p-d pairs with less amount of demand and adjusting the cycles with respect to the vessel capacities. The framework should be designed in such a way that at each iteration the solution obtained for the modified problem with equal vessel capacities is closer to a realistic solution that can be implemented. However, in this study, we only illustrate the solution of a MPDPSL-C with given demand info and equal capacities.

In this network problem, we have ports that can be both pickup and delivery points at the same time. In order to comprehend this structure clearly, let us consider a hypothetical cyclic route containing the ports A , B , C and D as shown in Figure 5.1, and let the quantity of p-d requests among these ports be as shown in Table 5.1. Let us denote the units of request from port i to port j by q_{ij} . For example, we have a request of $q_{AB} = 10$ units from port A to port B while we have a request of $q_{BA} = 7$ units from port B to port A . Even though we have only four ports in this cycle, we actually have eight p-d requests that are to be satisfied.

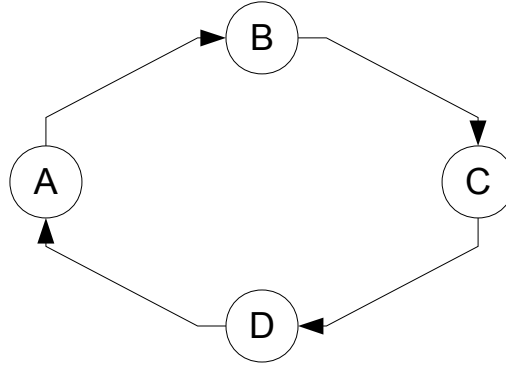


Figure 5.1: An example for a cyclic route with four ports

In the example in Figure 5.1, the vessel starts from port A picking up $(q_{AB} + q_{AD}) = 15$ units of load and moves on to port B delivering $q_{AB} = 10$ units and picking up $(q_{BA} + q_{BC}) = 15$ units. Then, it moves on to port C delivering $q_{BC} = 8$ and picking up $(q_{CA} + q_{CD}) = 10$, arrives at port D delivering $(q_{AD} + q_{CD}) = 9$ and picking up $(q_{DB} + q_{DC}) = 11$. It finally completes the cycle at port A . Note that the requests for a previously visited port is carried on to the next cycle.

	A	B	C	D
A	-	10	-	5
B	7	-	8	-
C	6	-	-	4
D	-	6	5	-

Table 5.1: p-d requests for the cycle in Figure 5.1

Generating the MPDPSL-C representation of this cyclic route is an easy task. For the cycle in Figure 5.1 and requests in Table 5.1, we create the MPDPSL-C representation shown in Table 5.2.

The cycle in Figure 5.1 is also equivalent to the MPDPSL-C representation in Figure 5.2 where each port is represented by multiple copies of the port each of which corresponds to a request associated with the port.

Pickup Point	Delivery Point	Request
p_{AB}	d_{AB}	q_{AB}
p_{AD}	d_{AD}	q_{AD}
p_{BA}	d_{BA}	q_{BA}
p_{BC}	d_{BC}	q_{BC}
p_{CA}	d_{CA}	q_{CA}
p_{CD}	d_{CD}	q_{CD}
p_{DB}	d_{DB}	q_{DB}
p_{DC}	d_{DC}	q_{DC}

Table 5.2: MPDPSL-C representation of the cycle in Figure 5.1

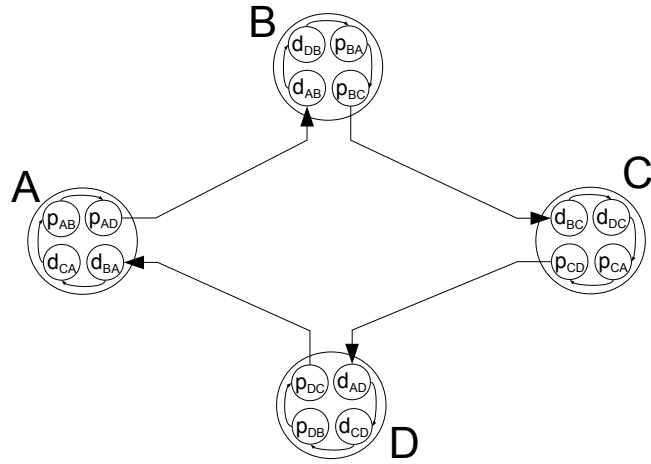


Figure 5.2: MPDPSL-C representation of the cycle in Figure 5.1

5.1 Modified TESA Algorithm for MPDPSL-C

As we create MPDPSL-C instances from the company's network, we have to reconcile the differences between MPDPSL and MPDPSL-C and modify our TESA Algorithm accordingly. Then, we will be able to solve a real life problem with real life data.

There are two main differences between the algorithmic implementations of MPDPSL and MPDPSL-C:

- the routes do not start and end at a depot location, and
- precedence constraints are not necessarily in effect.

The fact that routes do not start and end at a depot location does not have a significant effect on the principles of the TESA algorithm. However, lack of precedence constraints results in the absence of *blocks* in the routes. This renders the *Block Insert* and *Block Swap* phases ineffective. In MPDPSL, precedence constraints ensure that a route starts and ends with an empty vehicle, and we rely on this property while making use of *blocks* in the routes. In MPDPSL-C on the other hand, a vehicle may never be fully unloaded because the request of a port visited in one cycle may be carried over to the next cycle. Therefore, we may not utilize the *block* structure and need to eliminate the *Block Insert*, *Block Swap* and *Intra Route* phases from the algorithm. However, the *Insert/Split* phase, which is the core phase of the algorithm, works fine with the elimination of the precedence constraints and the depot location.

5.2 Computational Results

As we carry out a detailed parameter tuning for TESA algorithm on the *Insert/Split* phase parameters, we do not experiment with these parameters again and employ the same parameter settings for the modified TESA algorithm.

Employing the real life data provided to us by an international maritime transportation agency, we manage to solve an instance of MPDPSL-C with 122 p-d pairs distributed among 27 international ports. The demand range is set between 50 and 1500 containers and we assume all vessels have a capacity of 1500 containers. The maximum distance allowed per vessel is set to 7680 nautical miles assuming that a vessel can voyage 20 days a month for 24 hours a day with 16 knots/hour. The computational time for this instance is close to 2 minutes per restart. The cycle lengths, excess capacity for the cycles and the number of ports visited in the cycles for this MPDPSL-C instance are provided in Table 5.3. The solution has 16 vessels and an example of a cycle with five ports from the solution is given in Figure 5.3. Note that the average cycle length is close to 3650 knots, which takes almost two weeks of voyage but the demand is calculated for one month. In the solution framework, we have to adjust the demand and the cycle lengths; and, we have to ensure that we can accommodate the demand by the vessels available.

Cycle	Cycle Length (knots)	Excess Capacity (containers)	Number of Ports
1	2860	115	6
2	2441	1	4
3	3323	0	5
4	2706	8	4
5	4253	11	8
6	4298	17	6
7	3601	106	6
8	3807	0	5
9	3738	0	8
10	2951	0	5
11	3194	36	4
12	4461	7	9
13	2224	112	5
14	3656	9	6
15	5224	0	9
16	5369	3	10
Average	3631.63	26.56	6.25

Table 5.3: Results from an MPDPSL-C instance

At the current stage, we have shown that an MPDPSL-C instance can be solved with some minor modifications on our original TESA algorithm. As a result, this part of the study only prepares the background and necessary tools for the larger algorithmic framework that is to be designed for the solution of a real life MPDPSL-C instance.

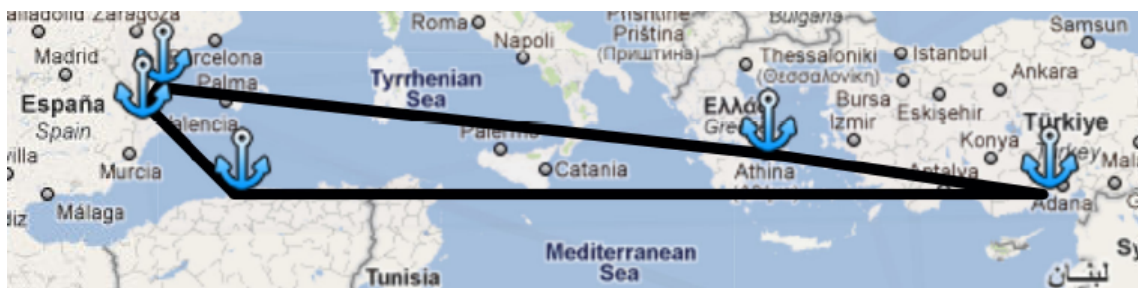


Figure 5.3: A cyclic route from the real life instance of MPDPSL-C

Chapter 6

Concluding Remarks

In this study, we present an efficient heuristic for MPDPSL. To the best of our knowledge, this is the first algorithm in the literature for the multi-vehicle version of the problem. To solve this problem, we develop a meta-heuristic algorithm where we combine the strengths of Tabu Search and Simulated Annealing to obtain an algorithm that exhibits both intensification and diversification capabilities. Due to the many possible ways of splitting a load among vehicle routes, search neighborhoods for MPDPSL are of high computational complexity. In our heuristic algorithm implementation, we employ a binary heap to search the neighborhoods efficiently and avoid excessive computation times.

This work also presents two sets of test problems that are obtained by modifications to test instances for similar problems in the literature. Results on these problems can serve as benchmark for future research on MPDPSL. Since no computational results are available on MPDPSL, we compare the solution quality of our heuristic with Nowak et al's [10] results on the single vehicle version. Even though our heuristic is not designed for this version, we demonstrate that it is capable of producing comparable and mostly higher quality solutions in comparable CPU time. In the absence of any benchmark results on the second problem set derived from the Ropke and Pisinger [13] instances, we present the first results and explore the benefit of split loads on this new set which has different characteristics than the first one. We observe that load splitting provides significant benefits on these instances as well; however, the benefits are relatively small compared to the first problem set. We attribute the difference to the spatial distribution of the pickup and delivery points in the two problem sets. Based on our observations on the two problem sets, we conclude that both load size range and spatial distribution of the pickup and delivery points are important factors in the magnitude of benefits that can be obtained from

split loads. Our study expands the findings of Nowak et al. on the benefit of split loads to the multiple vehicle version of the problem. We hope that these results draw attention to the potential savings that can be achieved by allowing load splits in many application areas of PDP.

Outsourcing, as an idea, has been around for a long time. However, in contemporary business practices, even third party logistics operators outsource some of their operations to a fourth party. In that respect, we define the MPDPSL with Outsourcing Option and modify our existing algorithm to obtain results for a problem set in the literature. We observe that the outsourcing option has significant potential based on the experiments we conduct on various cost schemes. We claim that when the pricing scheme of the company is different than that of the outside carrier, benefits obtained from outsourcing are significantly higher than when the pricing schemes of the company and the outside carrier are similar to each other.

We had a chance to observe the transportation network of an international maritime transportation agency and based on their network, we present MPDPSL with cyclic routes without depot. We modify our existing algorithm to match the structures of MPDPSL-C and present results on a real life case. In the future, we hope to utilize our MPDPSL-C algorithm in a larger framework to solve a real life problem with a few additional constraints that we have not discussed in this study.

While the proposed heuristic offers promising results, it is still worthwhile to explore exact solution approaches for the solution of MPDPSL. Experience from earlier work on similar problems suggests that exact approaches will be limited to producing optimal solutions for only very small size problems. Still, these solutions can serve as benchmark results to evaluate the performance of heuristics. Moreover, ideas from exact solution approaches can be utilized to build effective heuristics and lower bounds for the same problem. This is an avenue of research that we hope to pursue in the future.

Bibliography

- [1] C. Archetti, M. W. P Savelsbergh, and M. F. Speranza. To split or not to split: That is the question. *Transportation Research Part E*, 44:114–123, 2008.
- [2] C. Archetti and M. G. Speranza. The split delivery vehicle routing problem: A survey. In Ramesh Sharda, Stefan Voß, Bruce Golden, S. Raghavan, and Edward Wasil, editors, *The Vehicle Routing Problem: Latest Advances and New Challenges*, volume 43 of *Operations Research/Computer Science Interfaces Series*, pages 103–122. Springer US, 2008.
- [3] G. Clarke and J. W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12(4):568–581, 1964.
- [4] J-F. Cordeau, G. Laporte, and S. Ropke. Recent models and algorithms for one-to-one pickup and delivery problems. In Ramesh Sharda, Stefan Voß, Bruce Golden, S. Raghavan, and Edward Wasil, editors, *The Vehicle Routing Problem: Latest Advances and New Challenges*, volume 43 of *Operations Research/Computer Science Interfaces Series*, pages 327–357. Springer US, 2008.
- [5] J. J. Dongarra. Performance of various computers using standard linear equations software, (linpack benchmark report). *University of Tennessee Computer Science Technical Report*, 2010.
- [6] M. Dror and P. Trudeau. Savings by split delivery routing. *Transportation Science*, 23(2):141–145, 1989.
- [7] I. Gribkovskaia and G. Laporte. One-to-many-to-one single vehicle pickup and delivery problems. In Ramesh Sharda, Stefan Voß, Bruce Golden, S. Raghavan, and Edward Wasil, editors, *The Vehicle Routing Problem: Latest Advances and New Challenges*, volume 43 of *Operations Research/Computer Science Interfaces Series*, pages 359–377. Springer US, 2008.

- [8] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of vehicle routing and scheduling problems. *Networks*, 11(2), 1981.
- [9] H. Li and A. Lim. Local search with annealing-like restarts to solve the VRPTW. *European Journal of Operational Research*, 150(1):115–127, 2003.
- [10] M. Nowak, O. Ergun, and C. C. White III. Pickup and delivery with split loads. *Transportation Science*, 42(1):32–43, 2008.
- [11] M. Nowak, O. Ergun, and C. C. White III. An empirical study on the benefit of split loads with the pickup and delivery problem. *European Journal of Operational Research*, 198(3):734–740, 2009.
- [12] I. H. Osman. Heuristics for the generalised assignment problem: simulated annealing and tabu search approaches. *OR Spectrum*, 17:211–225, 1995.
- [13] S. Ropke and D. Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40:455–472, 2006.
- [14] S. R. Thangiah, I. H. Osman, and T. Sun. Hybrid genetic algorithm simulated annealing and tabu search methods for vehicle routing problem with time windows. *Technical Report 27, Computer Science Department, Slippery Rock University*, 1994.

Appendices

Appendix A

Algorithm Description and Computational Complexity Analysis

A.1 Notation

A *pair* (p, d) consists of a pickup stop and the corresponding delivery stop with indices p and d . The (possibly partial) load quantity of pair (p, d) is q_p . $s(p, d)$ denotes the decrease in route length obtained by removing the pair (p, d) from its current route. A *solution* $S = \{R_1, R_2, \dots, R_{|S|}\}$ is a collection of routes R that consist of pickup and delivery stops. $c(R)$ denotes the cost of route R and $c(S)$ denote the cost of the solution S . The tuple $\langle \ell_1^p, \ell_2^d \rangle - \langle \ell_2^p, \ell_2^d \rangle$ denotes the new pickup and delivery positions for pair (p, d) to be split into two such that part of the load is picked up and delivered after positions ℓ_1^p and ℓ_1^d , respectively and the remainder is picked up and delivered after positions ℓ_2^p and ℓ_2^d , respectively. If the pair (p, d) is inserted rather than split, then the pickup stop is inserted after ℓ_1^p , the delivery stop is inserted after ℓ_1^d and $\langle \ell_2^p, \ell_2^d \rangle = \emptyset$. $c(\langle \ell^p, \ell^d \rangle)$ is the cost of inserting p and d after the position ℓ^p and ℓ^d , respectively. Note that by triangular inequality, $s(p, d)$ and $c(\langle \ell^p, \ell^d \rangle)$ are always nonnegative. Finally, $r(\langle \ell^p, \ell^d \rangle)$ is the residual vehicle capacity for inserting p and d after position ℓ^p and ℓ^d , respectively.

A.2 Software Architecture and Data Structure

TESA Algorithm is implemented in C++ programming language. There are four main *classes* in the software architecture, namely *Stop*, *Block*, *Route* and *Solution*. *Stop* class consists of several attributes like *index*, *demand* et cetera and corresponds to a single *node*

of a p-d pair in the network. In *Block*, there are several *Stop* objects constituting a route segment. *Route* class consists of *Stop* objects and *Solution* class consists of *Route* objects. There are also several functions in these classes in order to insert/delete a *Stop* to/from a *Route*, calculate the objective function value and access/update the attributes of a *Stop* object.

The *Stops* in a *Route* are stored in a *doubly linked-list* which allows us to insert/delete in $O(1)$ time. Since accessing an element in a *linked-list* is in $O(|R|)$, where $|R|$ denotes the size of the route, we came up with a structure that allows us to access any *Stop* in a *Route* in $O(1)$. In order to access, delete or insert an element in a *linked-list* in C++, one should use a special *pointer* called an *iterator*. An *iterator* points to the corresponding element in the list and by incrementing the *iterator*, one can access the next element. Therefore, in order to find an element, we should scan the list starting from the first element, which can take up to $O(|R|)$. However, if we somehow store the *iterator* of an element then we can access to that element without scanning the whole list. Since the number of splits allowed for a *Stop* is limited, we have an upper bound on the maximum number of *Stops* in a *Solution*, which is $2nk$. Consequently, we can create an array of *iterators* at the size of this upper bound and access any *iterator* by using the *index* of the corresponding *Stop*. Accessing an element in an array by index is in $O(1)$ and accessing an element in a list by using an *iterator* is also in $O(1)$, so accessing an element in the list by index is in $O(1) + O(1) = O(1)$. Note that this structure is the equivalent of a *hash table* where the node indices are used instead of a *hash function*.

We also implemented a *Binary Heap* in order to improve the average running time of the *Insert/Split* phase. The function *makeHeap()* is used for creating a new heap H . Let *minimum(H)* denotes the function returning the pointer of the element with the minimum *key* value. Also, let *insert(H, x)* and *extractMin(H)* denote the functions for inserting the element x to H and deleting the element with the minimum *key* value while returning the pointer of this element.

A.3 Insert/Split Phase

The detailed pseudocode for selecting a candidate move for a given pair (p, d) is provided in Algorithm 5. Note that *minInsert(T)* is a function that returns the tuple $\langle \ell^p, \ell^d \rangle - \emptyset$ where $\langle \ell^p, \ell^d \rangle$ is the best position where entire load q_p can be inserted and function *minSplit(T)* returns the best two split positions $\langle \ell_1^p, \ell_2^d \rangle - \langle \ell_2^p, \ell_2^d \rangle$ where load q_p can be served in two partial shipments on the list T . The time complexity proofs for the worst

Algorithm 5 Insert/Split

```
1: Input:  $S, (p, d), Pr$  //S: Solution, (p,d): Selected pair
2: Output:  $\langle \ell_1^p, \ell_2^d \rangle - \langle \ell_2^p, \ell_2^d \rangle$  //Best position for insert or split
3:  $H \leftarrow \text{makeHeap}()$  //H: Binary heap of  $\langle \ell^p, \ell^d \rangle$  with key  $c(\langle \ell^p, \ell^d \rangle)$ 
4:  $T \leftarrow \emptyset$  //T: An array of  $\langle \ell^p, \ell^d \rangle$ 
5:  $SimList \leftarrow \emptyset$  //SimList: An array of  $\langle \ell_1^p, \ell_2^d \rangle - \langle \ell_2^p, \ell_2^d \rangle$ 
6:  $ChosenMove \leftarrow \emptyset$  //Initialize chosen move
7:  $isMoveChosen \leftarrow false$ 
8: for all  $R_i \in S$  do
9:   for all  $r(\langle \ell^p, \ell^d \rangle) > 0 \in R_i$  do
10:    if  $c(\langle \ell^p, \ell^d \rangle) + c(R_i) \leq D$  then
11:       $insert(H, \langle \ell^p, \ell^d \rangle)$ 
12:    end if
13:  end for
14: end for
15: while  $H \neq \emptyset$  and  $|SimList| \leq N_{move}$  do
16:   if  $r(\text{minimum}(H)) \geq q_p$  and  $\langle \ell^p, \ell^d \rangle$  is not TABU then
17:      $SimList \leftarrow SimList \cup \text{extractMin}(H)$ 
18:   else
19:     for all  $\langle \ell^p, \ell^d \rangle \in T$  do
20:      if  $r(\text{minimum}(H)) + r(\langle \ell^p, \ell^d \rangle) \geq q_p$  and  $\text{minimum}(H) - \langle \ell^p, \ell^d \rangle$ 
is not TABU and  $c(\text{bestSplit}(SimList)) > c(\text{minimum}(H)) + c(\langle \ell^p, \ell^d \rangle)$ 
then
21:         $SimList \leftarrow SimList \cup (\text{minimum}(H) - \langle \ell^p, \ell^d \rangle)$ 
22:      end if
23:    end for
24:     $insert(T, \text{extractMin}(H))$ 
25:  end if
26: end while
27:  $temp \leftarrow \text{CalculateTemperature}(Pr)$ 
28:  $ChosenMove \leftarrow \text{bestMove}(SimList)$ 
29: while  $isMoveChosen = false$  do
30:   Extract a  $\langle \ell_1^p, \ell_2^d \rangle - \langle \ell_2^p, \ell_2^d \rangle$  randomly from  $SimList$ 
31:    $prob \leftarrow \text{generateProbability}()$  //Generate a number  $\in (0, 1)$ 
32:    $\Delta C \leftarrow c(\langle \ell_1^p, \ell_2^d \rangle - \langle \ell_2^p, \ell_2^d \rangle) - s(\langle \ell_1^p, \ell_2^d \rangle - \langle \ell_2^p, \ell_2^d \rangle)$ 
33:   if  $prob \leq e^{-\Delta C / temp}$  then
34:      $ChosenMove \leftarrow \langle \ell_1^p, \ell_2^d \rangle - \langle \ell_2^p, \ell_2^d \rangle$ 
35:      $isMoveChosen = true$ 
36:   end if
37: end while
38: return  $ChosenMove$ 
```

and best cases of this phase are as follows.

Proposition 1 *The worst case running time of the Insert/Split Phase is in $O(n^5)$ and the best case running time is in $\Omega(n^3)$.*

Proof.

Worst Case Analysis. Since there are n pairs, the upper bound on the number of *stops* is $2nk$, where k denotes the maximum number of splits allowed for one p-d pair. Between lines 8-14 of Algorithm 5, we check for all possible positions $\langle \ell^p, \ell^d \rangle$ such that ℓ^p comes before ℓ^d , which takes exactly

$$|R_i| + (|R_i| - 1) + \dots + 1 = \frac{|R_i|(|R_i| + 1)}{2} \quad \forall R_i \in S$$

operations for one route. Since one operation is in $O(1)$, for one route the complexity is in $\Theta(|R_i|^2)$. For a particular solution S , i.e. $\forall R_i \in S$, it takes

$$\sum_{R_i \in S} \Theta(|R_i|^2)$$

which is in $O(n^2)$, since

$$\sum_{R_i \in S} |R_i| \leq 2nk \quad \text{and} \quad \sum_{R_i \in S} |R_i|^2 \leq \left(\sum_{R_i \in S} |R_i| \right)^2 \leq (2nk)^2$$

In the worst case, for each loop between 9-13, we insert an element to the Binary Heap created at the beginning of the every iteration. Since inserting an element to a heap is in $O(\log |H|)$, the entire loop takes up to $O(n^2 \log n^2) = O(n^2 2 \log n) = O(n^2 \log n)$.

The loop between 15-26 continues until the heap is empty. Since the size of the heap is in $O(n^2)$ and there is an inner loop between 19-23 that compares each element in the array T with the current $\text{minimum}(H)$, during the entire loop, $O(n^2 + (n^2 - 1) + \dots + 1) = O(n^4)$ comparisons are made. Each comparison takes $O(1)$ and $\text{minimum}(H)$ also takes $O(1)$. Also, there is an $\text{extractMin}(H)$ operation at each iteration, which takes $O(\log |H|)$. Therefore, the entire loop takes up to $O(n^2 \log n^2 + n^4) = O(n^4)$. However, this is only for one selected pair (p,d), since there are $O(n)$ pairs to be considered, the worst case running time of the *Insert/Split Phase* is in $O(n)(O(n^2 \log n) + O(n^4)) = O(n^5)$.

Best Case Analysis. For the loop between 8-14, we still have to examine $O(n^2)$ positions $\langle \ell^p, \ell^d \rangle$, however, due to the distance and capacity constraints, there might not

be $O(n^2)$ elements in the heap. In fact, it is possible that there is no element in the heap, which makes the loop between 15-26 run in $\Omega(1)$. Again, there are $O(n)$ pairs to consider, so, in the best case, the running time of the *Insert/Split Phase* is in $O(n)\Omega(n^2) = \Omega(n^3)$.

■

Even though we have a worst case running time of $O(n^5)$, computational experiments showed that on average, there are $O(n)$ elements in the heap, which makes the average running time of the phase $O(n)(O(n^2 \log n) + O(n^2)) = O(n^3 \log n)$.

A.4 Block Insert Phase

As discussed before, a *block* is a route segment starting and ending with an empty vehicle, there might be several *blocks* in a route or sometimes a route itself might be a single *block*. In one iteration of this phase, a *block* B^* is selected from the solution and inserted to another location ℓ^{B^*} such that $s(B^*) - c(\ell^{B^*})$ is largest. Since the capacity and the precedence constraints remain intact between *block* movements, only the route length constraint is taken into consideration. A pseudocode for *BlockInsert*(S_c) is provided in Algorithm 6.

Algorithm 6 BlockInsert

```

1: Input:  $S_c$  // S: Solution
2: Output:  $B^*, \ell^{B^*}$  // Selected block and the best position
3:  $B^* \leftarrow \emptyset$  // Initialize selected block
4:  $\ell^{B^*} \leftarrow \emptyset$  // Initialize best position
5: for all  $B \in S_c$  do
6:   for all  $\ell^B \in S_c$  do
7:     if  $c(\ell^B) + c(R_{\ell^B}) \leq D$  then
8:       if  $c(\ell^B) - s(B) < c(\ell^{B^*}) - s(B^*)$  then
9:          $B^* \leftarrow B$ 
10:         $\ell^{B^*} \leftarrow \ell^B$ 
11:       end if
12:     end if
13:   end for
14: end for
15: return  $B^*, \ell^{B^*}$ 

```

A.5 Block Swap Operator

In one iteration of this phase, two blocks $B' - B''$ are selected and they are inserted into each other's routes such that $(s(B') + s(B'')) - (c(\ell^{B'}) + c(\ell^{B''}))$ is largest. Note that $\ell^{B'}$ and $\ell^{B''}$ denote the best possible locations in these routes for B' and B'' respectively. A pseudocode for $BlockSwap(S_c)$ is provided in Algorithm 7.

Algorithm 7 BlockSwap

```

1: Input:  $S_c$  // S: Solution
2: Output:  $B', B'', \ell^{B'}, \ell^{B''}$  // Selected blocks and the best positions
3:  $B' \leftarrow \emptyset$  // Initialize selected block
4:  $B'' \leftarrow \emptyset$  // Initialize selected block
5:  $\ell^{B'} \leftarrow \emptyset$  // Initialize best position
6:  $\ell^{B''} \leftarrow \emptyset$  // Initialize best position
7: for all  $B_i \in S_c$  do
8:   for all  $B_j, i \neq j \in S_c$  do
9:     for all  $\ell^{B_i} \in R_{B_j}$  do
10:      if  $c(\ell^{B_i}) + c(R_{B_j}) \leq D$  then
11:        if  $c(\ell^{B_i}) - s(B_i) < c(\ell^{B'}) - s(B')$  then
12:           $B' \leftarrow B_i$ 
13:           $\ell^{B'} \leftarrow \ell^{B_i}$ 
14:        end if
15:      end if
16:    end for
17:    for all  $\ell^{B_j} \in R_{B_i}$  do
18:      if  $c(\ell^{B_j}) + c(R_{B_i}) \leq D$  then
19:        if  $c(\ell^{B_j}) - s(B_j) < c(\ell^{B''}) - s(B'')$  then
20:           $B'' \leftarrow B_j$ 
21:           $\ell^{B''} \leftarrow \ell^{B_j}$ 
22:        end if
23:      end if
24:    end for
25:  end for
26: end for
27: return  $B', B'', \ell^{B'}, \ell^{B''}$ 

```

Appendix B

Results of Ropke and Pisinger Problem Instances

B.1 Split vs. No-split Solutions

Load Range Instance	0.25-1				0.51-0.60			
	Initial	Split	No-split	Imp. (%)	Initial	Split	No-split	Imp. (%)
A	17621.60	15899.60	15973.60	0.46	17539.60	16791.20	16926.50	0.80
B	18404.10	16351.30	16327.10	-0.15	17123.00	17115.50	17076.30	-0.23
C	15707.80	14475.70	14486.50	0.07	15051.00	14956.00	15000.60	0.30
D	16779.30	15367.60	15348.40	-0.13	16546.00	16290.00	16248.60	-0.25
E	12047.50	11118.10	11191.90	0.66	12423.00	11397.50	12149.70	6.19
F	12463.40	9979.03	10013.70	0.35	11026.30	9532.59	10779.90	11.57
G	10814.60	9876.62	9902.91	0.27	10875.00	9665.06	10742.60	10.03
H	9215.82	8670.63	8770.05	1.13	9218.57	9199.58	9202.81	0.04
I	15875.90	13241.60	13448.50	1.54	14935.80	14469.40	14592.70	0.84
J	14671.50	12927.10	12957.00	0.23	13576.30	13200.20	13201.00	0.01
K	14411.40	12685.20	12762.50	0.61	13097.80	12759.30	13086.90	2.50
L	15561.00	13958.00	14067.80	0.78	15727.10	14867.80	14919.90	0.35
Average				0.48				2.67

Table B.1: Results of the problems in Ropke and Pisinger [13] with 50 nodes comparing the tour length of split solutions versus no-split solutions.

Load Range	0.25-1				0.51-0.60			
Instance	Initial	Split	No-split	Imp. (%)	Initial	Split	No-split	Imp. (%)
A	31003.40	26274.60	26182.10	-0.35	28913.30	27301.20	28122.90	2.92
B	31383.90	25395.60	25618.00	0.87	28466.60	27090.10	28289.20	4.24
C	31959.80	25876.20	25919.30	0.17	27925.80	27221.30	27657.50	1.58
D	31939.40	28295.30	28606.70	1.09	30084.10	28574.70	29725.00	3.87
E	18055.40	16102.10	16361.50	1.59	17595.20	15320.00	17549.00	12.70
F	19714.40	15702.40	15691.70	-0.07	18159.30	17574.20	18071.70	2.75
G	19964.10	14756.20	15104.70	2.31	17884.60	14888.40	17709.20	15.93
H	21055.10	16711.90	17291.90	3.35	19042.60	16259.70	19017.50	14.50
I	28674.20	24899.10	25235.30	1.33	27931.70	24994.40	27288.60	8.41
J	25719.10	22611.80	22655.70	0.19	24220.30	23025.50	23517.60	2.09
K	31055.50	23187.30	23440.20	1.08	26920.40	24509.00	26331.20	6.92
L	26584.20	21961.90	22172.10	0.95	24795.60	23994.70	24100.30	0.44
Average				1.04				6.36

Table B.2: Results of the problems in Ropke and Pisinger [13] with 100 nodes comparing the tour length of split solutions versus no-split solutions.

Load Range	0.25-1				0.51-0.60			
Instance	Initial	Split	No-split	Imp. (%)	Initial	Split	No-split	Imp. (%)
A	68806.60	57872.80	58704.50	1.42	64657.40	58847.60	64091.80	8.18
B	65680.40	55707.90	56200.40	0.88	63230.90	57559.10	62719.00	8.23
C	66912.80	57003.80	57681.00	1.17	63353.50	57495.90	62594.90	8.15
D	66896.50	58744.30	59335.40	1.00	64698.60	59396.70	64426.30	7.81
E	41316.30	30138.20	30502.30	1.19	37890.80	31736.80	37506.20	15.38
F	36325.20	29185.90	30153.70	3.21	33552.50	27596.00	33192.00	16.86
G	38698.30	30068.50	30626.50	1.82	34876.50	29421.80	34632.20	15.04
H	38990.90	30414.00	31320.20	2.89	36151.50	31911.50	35752.20	10.74
I	62167.30	50830.60	51338.80	0.99	57721.30	50154.80	56830.80	11.75
J	67448.80	54355.60	55120.80	1.39	61133.20	53636.20	60668.30	11.59
K	62832.40	51120.50	51949.00	1.59	57895.90	50084.40	57012.90	12.15
L	67103.90	53828.60	54631.40	1.47	63662.70	54393.40	62504.20	12.98
Average				1.58				11.57

Table B.3: Results of the problems in Ropke and Pisinger [13] with 250 nodes comparing the tour length of split solutions versus no-split solutions.

Load Range	0.25-1				0.51-0.60			
Instance	Initial	Split	No-split	Imp. (%)	Initial	Split	No-split	Imp. (%)
A	122227.00	107933.00	109432.00	1.37	117079.00	106674.00	116736.00	8.62
B	130599.00	114731.00	116553.00	1.56	124167.00	110881.00	123336.00	10.10
C	130003.00	110639.00	111894.00	1.12	120185.00	109181.00	119672.00	8.77
D	131867.00	112940.00	114494.00	1.36	121301.00	109746.00	120482.00	8.91
E	90946.60	67132.20	70243.70	4.43	82358.30	63068.40	81880.80	22.98
F	97917.70	73073.30	75601.10	3.34	90732.80	68829.70	89849.90	23.39
G	99146.70	72823.80	76561.80	4.88	91885.90	70038.80	91386.90	23.36
H	82228.20	62928.10	65960.10	4.60	76734.40	60568.50	76288.90	20.61
I	115321.00	94473.60	97124.40	2.73	108329.00	93178.20	107403.00	13.24
J	123209.00	101738.00	103359.00	1.57	113252.00	96984.80	112239.00	13.59
K	121653.00	103395.00	105728.00	2.21	114033.00	97429.50	113463.00	14.13
L	120380.00	102316.00	104367.00	1.97	115473.00	98102.70	114510.00	14.33
Average				2.59				15.16

Table B.4: Results of the problems in Ropke and Pisinger [13] with 500 nodes comparing the tour length of split solutions versus no-split solutions.

B.2 Computational Times

Table B.5: CPU time per restart (in seconds) to solve the problems when the load size range is 0.25-1.

Number of requests	50		100		250		500	
Problem	Split	No-split	Split	No-split	Split	No-split	Split	No-split
A	3.4	2.8	19.9	17.4	202.1	186.7	673.8	661.4
B	2.9	2.3	18.9	17.2	189.9	165.2	620.0	582.2
C	3.3	2.8	22.5	18.6	202.3	191.6	603.2	615.6
D	2.8	2.3	20.2	18.4	184.8	178.1	638.2	634.0
E	5.5	4.1	44.8	41.2	658.6	359.2	1286.4	1061.2
F	4.0	2.0	51.8	25.6	815.6	505.0	1293.0	1101.2
G	4.8	3.4	34.3	30.7	621.8	396.6	1221.6	1200.6
H	6.1	4.8	41.9	39.2	849.8	414.5	1876.0	1915.0
I	3.5	2.5	29.6	22.5	297.7	236.2	985.2	925.8
J	5.0	4.4	31.5	25.0	298.0	234.0	811.2	796.8
K	3.3	2.8	19.5	14.3	348.6	285.4	751.0	778.8
L	2.7	2.1	32.1	19.2	255.1	179.7	870.8	778.6
Average	3.9	3.0	30.6	24.1	410.3	277.7	969.2	920.9

Table B.6: CPU time per restart (in seconds) to solve the problems when the load size range is 0.51-0.60.

Number of requests	50		100		250		500	
Problem	Split	No-split	Split	No-split	Split	No-split	Split	No-split
A	4.5	4.2	25.1	31.8	287.3	320.7	2124.6	954.8
B	4.3	4.5	19.4	34.4	253.0	314.4	2374.2	718.4
C	4.5	5.0	34.0	37.8	299.5	323.1	1985.8	1144.4
D	4.3	4.0	19.0	28.8	356.1	344.5	2247.0	849.4
E	7.6	6.3	74.7	56.4	3174.6	868.9	10860.4	2090.6
F	7.4	6.3	95.8	70.1	1123.1	912.7	10815.8	2051.4
G	6.2	7.6	50.1	50.9	1089.3	1672.8	11101.8	1604.0
H	11.2	9.1	57.9	67.5	939.5	704.5	16763.8	2586.0
I	5.8	5.3	32.4	38.0	468.2	322.0	5075.4	1485.6
J	6.5	5.4	37.5	46.2	448.8	345.2	4698.0	1183.6
K	2.3	5.5	30.4	40.4	537.5	325.4	4539.6	1250.8
L	4.4	3.9	49.3	35.1	392.3	372.5	5996.2	1241.0
Average	5.7	5.6	43.8	44.8	780.8	568.9	6548.6	1430.0