Solving the Precedence Constrained Vehicle Routing Problem with Time Windows Using the Reactive Tabu Search Metastrategy

by

William Paul Nanry, B.S., M.A.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May, 1998

19980721 015

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

DTIC QUALITY INSPECTED 3

Solving the Precedence Constrained Vehicle

Routing Problem with Time Windows

Using the Reactive Tabu Search Metastrategy

Approved by
Dissertation Committee:

Wesley Barnes

D. H. Morton

Valerie Tardif

Paul A. Jensen

Uttarayan Bagchi

John Chambers

Dedication

In faithful service to my Lord and Savior, Jesus Christ

I will extol the Lord at all times; his praise will always be on my lips.
My soul will boast in the Lord; let the afflicted hear and rejoice.
Glorify the Lord with me; let us exalt his name together.
I sought the Lord, and he answered me; he delivered me from all my fears.

Psalm 34:1-4

Acknowledgments

I want to convey my deepest appreciation to my family for their untiring support and love while I completed my dissertation. Dori, Chris and Bill contributed more to this effort than they will ever know. Their love and devotion truly helped preserve my sanity during the stressful times of this assignment.

I especially want to thank my supervisor, Professor J. Wesley Barnes, for his assistance. Professor Barnes patiently mentored me, helping me to set tangible goals so I would not become overwhelmed by the research. He enthusiastically responded to all my questions and motivated me to explore new avenues in my research. His devotion, expert advice and effective guidance greatly assisted me in completing this research.

I finally wish to thank the members of my dissertation committee - Drs. Paul A. Jensen, David P. Morton, Valerie Tardif, Uttarayan Bagchi and John Chambers. I appreciated their willingness to make my research effort a truly worthwhile process. Their professional expertise made my research a challenging and rewarding experience. They set an example of professional excellence I hope to emulate.

William P. Nanry

Austin, TX
May 15, 1998

Solving the Precedence Constrained Vehicle

Routing Problem with Time Windows

Using the Reactive Tabu Search Metastrategy

Publication No._____

William Paul Nanry, Ph.D.

The University of Texas at Austin, 1998

Supervisor: J. Wesley Barnes.

The vehicle routing problem (VRP) is associated with the design of a set of minimum cost routes for a fleet of vehicles to serve, exactly once, a set of customers with known demands. The pickup and delivery problem with time windows (PDPTW) is a generalization of the VRP. The PDPTW constructs optimal routes to satisfy transportation requests, each requiring both pickup and delivery under capacity, time window, precedence and coupling constraints. This dissertation presents a reactive tabu search (RTS) approach to solve the PDPTW and illustrates how to transform generalized precedence constrained routing problems with time windows (PCRPTW) into equivalent PDPTWs.

The PDPTW algorithm uses three distinct search neighborhoods that capitalize on the dominance of the precedence and coupling constraints. The algorithm employs a multineighborhood strategic search methodology, to alternate between search neighborhoods in order to negotiate different regions of the solution space and alter directions of search. All tours generated by the search require that suppliers and the corresponding deliveries be located on the same route and ordered properly. Because RTS explores infeasible solutions, it can discover time window or load infeasible solutions having significantly lower objective values than the optimal.

This dissertation also establishes benchmark data sets for the PDPTW through the modification of the input data structure for Solomon's (1987) benchmark VRPTW data sets. Computational results substantiate the solution quality and efficiency of the PDPTW algorithm when compared against one heuristic and two exact VRPTW algorithms.

Finally, this dissertation demonstrates how to transform representative *generalized*-PCRPTWs into PDPTWs. Modifications to the input data structure are presented and example problems are used to display the transformations.

Table of Contents

Dedication iv

Acknowledgments v

Abstract vi

List of Tables xii

List of Figures xv

Chapter 1: Introduction 1

1.1 The Time Windows Constrained Routing Problems 1

1.2 The search engine 4

1.3 Research Objectives 6

Chapter 2: Literature Review 9

2.1 The Vehicle Routing Problem with Time Windows 9

2.1.1 Exact Methods 10

2.1.2 Heuristic Methods 13

2.2 Pickup and Delivery and Dial-a-Ride Problems with Time Windows 21

2.2.1 Exact Methods 22

2.2.2 Heuristic Methods 25

2.3 Reactive Tabu Search (RTS) 27

2.3.1 Tabu Search Algorithms from the Literature 27

2.3.2 The Reactive Tabu Search (RTS) Heuristic 39

Chapter 3: Detailed Problem Description 42

3.1 The Advanced Request Problem 43

3.2 Objectives 43

3.3 Side Constraints 45

3.4 PDPTW Assumptions, Notation and Definitions 48

Chapter 4: The Algorithm 55

4.1 The Input Data and Initial Tour 55

4.1.1 The Benchmark Data Sets 55

4.1.2 The Input Data Structure 56

4.1.3 The Optimal Solution vs Best Solution Found 59

4.1.4 Infeasible Initial Tour 60

4.1.5 Feasible Initial Tour with a Vehicle Reduction Phase 62

4.1.6 Feasible Initial Tour without a Vehicle Reduction Phase 63

4.2 Neighborhood Search Strategies 63

4.2.1 Single Precedence Ordered Subset Insertion (SPI) 64

4.2.2 Swapping Pairs Between Routes 67

4.2.3 Within Route Insertion (WRI) 70

4.2.4 Limiting the Search 73

4.2.5 Time Windows Reduction 75

4.2.6 Inadmissible Arcs 77

4.3 Two-Level Open Hashing Structure 78

4.4 The Tabu Criteria, Length and Data Structures 79

4.5 The Algorithms 82

4.6 Computational Results 96

4.6.1 25-customer problems 98

4.6.2 50-customer problems 103

4.6.3 100-customer problems 114

4.7 Conclusions 120

Chapter 5: The Generalized Precedence Scenarios 123

5.1 The Single Supplier Supporting Several Delivery Locations 124

5.2 Several Suppliers Supporting a Single Delivery Location 134

5.3 The Serial Precedence Model 141

5.4 Conclusions 150

Chapter 6: Areas for Further Research and Summary 153

6.1 Areas for Further Investigation 153

6.2 Extensions to this Research 157

6.3 Major Contributions of this Research 160

6.4 Summary 162

Appendix A: Data Sets Examined 164

Appendix B: Example of the Modified Data Structure 168

Appendix C: The Code for the PDPTW Algorithm 173

Reference List 215

Vita

List of Tables

Table 1 - Effect of iterations on the search 87

Table 2: INIT results, 25-customers 99

Table 3: INIT infeasible results, 25-customers 100

Table 4: NEW results, 25-customers 101

Table 5: NEW infeasible results, 25-customers 102

Table 6: NEWc results, 25-customers 102

Table 7: NEWc infeasible results, 25-customers 103

Table 8 - 50-customer INIT results where optimal schedules are known .. 105

Table 9 - INIT results for remaining 50-customer problems 106

Table 10 - INIT results, 50-customers 107

Table 11 - Infeasible solutions using INIT 108

Table 12 - 50-customer NEW results where optimal schedules are known 108

Table 13 - NEW results for remaining 50-customer problems 109

Table 14 - NEW results, 50-customers 110

Table 15 - Infeasible tours using NEW 111

Table 16 - 50-customer NEWc results where optimal schedules are known 112

Table 17 - NEWc results for remaining 50-customer problems 112

Table 18 - NEWc results, 50-customers 114

Table 19 - NEWc infeasible results, 50-customers 114

Table 20 - 100-customer NEW results where optimal schedules are known 115

Table 21 - NEW results for remaining 100-customer problems 116

Table 22 - NEW results, 100-customers 117

Table 23 - 100-customer NEWc results where optimal schedules are known 118

Table 24 - NEWc results for remaining 100-customer problems 119

Table 25 - NEWc results, 100-customers 119

Table 26: The supporting input data structure for the single-to-many model 124

Table 27: The supporting input data structure for the transformed model . 125

Table 28: Optimal Tour Data for SINGLEC 129

Table 29: Optimal Tour Data for SINGLER 130

Table 30: Infeasible Tour Data for SINGLER 132

Table 31: Optimal Tour Data for SINGLEM 134

Table 32: Optimal Tour Data for Route 29 of MANYC 136

Table 33: Optimal Tour Data for MANYR 139

Table 34: Optimal Tour for MANYM 140

Table 35: Initial input data for the serial precedence model 141

Table 36: The transformed input data structure for the serial precedence model 142

Table 37: Optimal Tour Data for SERIAL1 144

Table 38: Optimal Tour Data for Route 32 of SERIAL2 145

Table 39: Optimal Tour Results for SERIAL3 147

Table 40: Infeasible Tour Results for SERIAL3 149

Table 41: Input Data for Generalized Scenario 151

Table 42: The Transformed Input Data Set 151

Table B.1: Original Solomon Data for 25-customer r110 170

Table B.2: Modified Data Structure for PDPTW - nr110 171

Table B.3: Optimal Tour for nr110 172

List of Figures

Figure 1: An Example of Single Precedence Ordered Subset Insertion ... 64

Figure 2: The Swap Pairs Neighborhood Search 68

Figure 3: The WRI Neighborhood Search 71

Figure 4: The generalized precedence digraph 124

Figure 5: The transformed digraph 125

Chapter 1

Introduction

Economic incidents such as the oil crisis of the early 1970's, deregulation of the U.S. airline and trucking industry in the 1980's and the rapidly declining military budget have motivated both private companies and academic researchers to vigorously pursue new methods to improve the efficiency of logistics distribution and transportation. This rekindled interest has fueled the recent development in metaheuristic procedures. New developments in adaptive memory strategies of tabu search (TS) have been especially productive in solving difficult combinatorial optimization problems with greater effectiveness than ever before. Time windows constrained routing problems form a large segment of this class of combinatorial optimization problems.

In this chapter we overview time window constrained routing problems, discuss the search engine and outline the major objectives of this research.

1.1 Time Windows Constrained Routing Problems

The vehicle routing problem (VRP) involves designing a set of minimum cost routes, beginning and terminating at a depot, for a fleet of vehicles which services a set of customers with known demands exactly once. The vehicles will

either deliver supplies to, or collect products from, customers along their designated routes. The number of customers that may be serviced by the vehicle is limited by both the vehicle's cargo capacity and by the time available to service the customers. The temporal aspect of these intrinsically spatial problems has become increasingly important as manufacturing, service and transportation companies have tried to not only cut their logistics costs, but also to provide superior service in competitive environments. The time dimension has been incorporated in these problems in the form of customer-imposed time window constraints (VRPTW) (Ball et al. 1995, 35). These constraints prohibit customer service starting prior to a prescribed earliest time and service may not begin after a specified latest time. This added complexity of allowable delivery time windows further complicates the vehicle routing problem.

The pickup and delivery problem with time windows (PDPTW) is a generalization of the VRPTW. The PDPTW requires satisfying a set of transportation requests, known in advance, by a homogeneous vehicle fleet housed at one depot. Each transportation request requires picking up material at a predetermined location during its associated time window and delivering it to a "paired" destination during its time window. Loading and unloading times are incurred at each location. In addition to the above precedence constraints, each

route must satisfy *coupling* constraints since paired pickup and delivery locations must be serviced by the same vehicle. Precedence and coupling constraints are viewed as "hard" constraints and will not be violated at any iteration. Since the load fluctuates depending on the sequencing of customers on a vehicle route, the planning horizon can be more restrictive to the PDPTW than the VRP. A further generalization of the PDPTW is the generalized precedence constrained routing problem with time windows (PCRPTW). The PCRPTW requires the design of optimal routes to satisfy transportation requests, each requiring pickup at one or several suppliers, where order for pickup may or may not be predetermined, with delivery to one or several destinations, again, where the order for delivery may or may not be preordained.

Precedence constrained routing problems arise under a variety of circumstances. They describe situations in which vehicles, aircraft, trucks, even people, must travel to a variety of places to deliver and/or pick up goods and provide services. Some practical applications include the dial-a-ride problem, airline scheduling, bus routing, tractor-trailer problems, helicopter support of offshore oil field platforms and logistics and maintenance support. They also arise in less obvious situations such as VLSI circuit design, flexible manufacturing systems and evacuating casualties.

One scenario where the solution of these types is important is in the operation of a fleet of supply vehicles. The enormous costs of acquisition or leasing of additional vehicles often compels managers to exhaust the capacity of the existing fleet while increasing fuel, maintenance and driver overtime costs pressure vehicle routes to be as short as possible.

Although time windows constrained routing problems may be simply stated they are usually extremely difficult to solve. The most basic problem considered in this research, the traveling salesman problem with time windows, is known to be NP-*complete* (Savelsbergh, 1985).

Motivation for the research presented in this dissertation is rooted in the operations of the Defense Logistics Agency (DLA). DLA operates a variety of transportation services and is responsible for coordinating and scheduling the logistics support for all three branches of the US Armed Services.

1.2 The Search Engine

Time window constraints usually complicate the search process and lead to a distinctly different solution space from instances with no TW constraints. TW constraints may partition the solution space into disjoint feasibility regions. This underlying structure defeats classical optimization techniques for all but the smallest and most tightly constrained instances. A very robust search engine is

required to quickly determine near optimal solutions to this type of problem. The associated search strategy must be able to traverse infeasible regions of the solution space while searching for excellent feasible solutions. The search neighborhood is restricted by precedence and coupling constraints and strong time windows feasibility conditions. These restrictions augment the search's efficiency by limiting the number of neighbor solutions that must be examined. Once an "elite" solution is identified, the search needs to "intensify" the search in the immediate vicinity to determine the possible presence of other elite solutions. When too many solutions are being revisited, the search trajectory needs to be altered to diversify and move the search into another region of the solution space. Reactive tabu search (RTS) meets all these requirements and has proven to be most effective in solving this class of combinatorial optimization problems (Glover, 1996).

One of the benefits of RTS is its ability to traverse infeasible regions of the solution space. While traversing infeasible regions, RTS often discovers infeasible solutions having significantly lower objective values than the optimum. These infeasible solutions can be valuable to managers desiring to improve overall system performance at a reduced cost to the company. Since violation of the precedence and coupling constraints are not allowed, infeasible

solutions will violate either time window or capacity constraints. Some time window violations are negligible and would not have to be coordinated with the customer. More significant time windows violations would have to be negotiated with customers. If the customer permits a late delivery, the manager could reduce the delivery charge to that customer while significantly lowering the company's distribution costs.

1.3 Research Objectives

The primary objective of this research is to develop an efficient and competitive solution approach to solve the pickup and delivery problem with time windows (PDPTW). Inherent in this objective is the understanding that this approach could handle diverse vehicle routing problem instances. Thus, this research will develop methods to transform several generalized precedence constrained routing problems with time windows (PCRPTW) into equivalent PDPTWs.

To accomplish this objective the RTS metaheuristic as developed by Battiti and Tecchioli (1994) will be employed. In contrast to simple local search techniques, RTS does not stop when local improvement is not possible. The best alternative in the current neighborhood is selected as the new solution, even if it is worse than the current solution. This strategy allows the method to escape a

local optimum and explore a larger portion of the solution space. Any solutions that would reverse the direction of the search by visiting recently visited moves are considered tabu.

While RTS incorporates the basic parameters and memory structures of tabu search, it dynamically adjusts the search parameters based on the quality of the search path. The tuning is automated and executed while the algorithm runs and monitors its past behavior. The quality of the search is determined by the number of iterations occurring since the last time a particular solution has been visited. High quality solution paths seldom revisit, or "cycle", to the same solution. If a solution is repeated, RTS adjusts the tabu length search parameter to discourage further repetitions. If numerous, high quality solutions are being identified, RTS intensifies the search in that solution subspace to determine the best solution in that locale. When too many solutions are repeated too often, the search is assumed to be trapped in a local attractor basin. RTS provides a diversification strategy that allows the search to overcome the "attraction" and climb out of the local attractor basin into other regions of the solution space.

A key issue to the effective implementation of RTS is the accurate identification of previously visited solutions. The two-level hashing scheme proposed by Horowitz, Sahni and Freed, 1993, and implemented by Carlton,

1995, will also be implemented in this research. Many of the initial ideas for this work, in addition to the two-level hashing, were taken from Carlton. The two-level hashing structure has proven to be effective in minimizing collisions, where two nonidentical solutions are incorrectly determined to be duplicate solutions.

Chapter 2
Literature Review

The body of literature that exists for the efficient routing and scheduling of transportation resources is overwhelming. Bodin et al. (1983) completed an exhaustive review of exact and heuristic methods for the family of vehicle routing problems. Solomon and Desrosiers (1988) survey solution techniques for time window constrained problems and furnish 80 references. More recently, Ball et al. (1995) provided an updated review for solving network routing problems.

This review will focus on exact and heuristic methods developed for solving the time constrained vehicle routing and pickup and delivery problems (VRPTW and PDPTW) and previous tabu search methods used to solve vehicle routing problems. The reader is referred to the reviews of Carlton (1995) and Desrosiers, Dumas, Solomon and Soumis (1995) for alternate time window constrained problems, like the traveling salesman problem with time windows (TSPTW) and the shortest path problem with time windows (SPPTW).

2.1 The Vehicle Routing Problem with Time Windows

The VRPTW consists of designing a set of minimum cost routes, originating and terminating at a common depot, for a fleet of vehicles serving a set of customers with known demands. In a feasible schedule, customers are

visited exactly once, no vehicle capacity is exceeded, and each customer service is provided within a specified time window.

The VRPTW generalizes the vehicle routing problem (VRP) by adding time windows. Since the VRP is NP-hard so is the VRPTW. Even finding a feasible solution to the VRPTW when the number of tours is fixed in advance is NP-complete. It is therefore unlikely that an efficient polynomial algorithm will be developed for the VRPTW.

Vehicle routing and scheduling problems with time window constraints are typically encountered where the customer must provide access, verification or payment upon delivery of the product or service. Bank deliveries, industrial refuse collection and school bus routing and scheduling provide a few representative examples of time constrained routing and scheduling problems.

2.1.1. Exact Methods

Desrochers, Lenstra, Savelsbergh and Soumis (1988) state that dynamic programming and branch-and-bound methods are the primary approaches considered for optimization of the VRPTW. Two preferred methods are set partitioning, which solves a continuous relaxation of the problem using column generation and the use of state space relaxation to compute lower bounds. They

conclude that optimization algorithms are unlikely to be able to solve large scale problems.

Their conclusions are supported by Desrochers, Desrosiers and Solomon (1992) who use a column generation scheme to solve the LP relaxation of the set partitioning formulation of the VRPTW. Instead of initially fixing the fleet size, they determine the minimal homogeneous fleet size simultaneously with the best sets of routes and schedules. Before the authors solve the problem, they reduce the time windows' widths to achieve a more tractable problem. Optimal solutions are found for seven out of the twenty-one 100-customer problem sets investigated. The consistently large computational cost incurred indicates a need to pursue heuristic methods.

Earlier, Desrosiers, Soumis and Desrochers (1984) employed a column generation scheme on a set partitioning problem solved with linear programming and branch-and-bound to solve the VRPTW. The columns were generated by using dynamic programming to solve a shortest path subproblem with time windows on the nodes. The problem is a generalization of the m-traveling salesman problem. The optimal solution found first minimizes the number of vehicles and for this number of vehicles, minimizes travel costs. Experimental

results are offered on six bus transportation problems along with numerous problem variants.

Kolen, Rinooy Kan and Trienkens (1987) use the branch-and-bound approach to solve the VRPTW for a fixed fleet of vehicles, with limited capacity, stationed at a common depot. The authors used a shortest path labeling method similar to Dijkstra's method. Their test problems were small, ranging from 6 to only 15 customers. They verified that the width and number of time window constraints significantly influenced the run time of their algorithm. As time windows become larger, more feasible solutions need to be examined which requires more time. The number of vehicles or capacity of the vehicles have much less influence on the computational results. Their study also reveals the difficulty in allowing vehicles to revisit a node.

The most recent and successful of the exact optimization methods was developed by Kohl (1995). Like Desrochers, Desrosiers and Solomon (1992), Kohl uses a decomposition method to solve the VRPTW. Kohl shows how the constrained shortest path problem (SPP) can be solved efficiently and presents several strategies for solving the coordinating master problem, primarily through the use of branch-and-bound methods. The lower bound on the optimal objective obtained from solving the SPP is improved further through the incorporation of

valid inequalities. This is the first application of valid inequalities on the VRPTW and marks a significant computational improvement in solving the VRPTW. Solutions to a large number of previously unresolved benchmark problems by Solomon (1987) are reported. Of the 87 benchmark problems 70 were solved to optimality. Desrochers, Desrosiers and Solomon were able to solve 50 of these benchmark problems.

2.1.2. Heuristic Methods

Since the cited optimal methods are effective only for relatively small problems or problems with a tailored structure, researchers have pursued heuristic approaches to solve the VRPTW. Some of the heuristic methods developed garner nearly optimal solutions in only a fraction of the computational cost.

Many route construction and improvement procedures have been developed. Route construction algorithms shape a feasible solution by inserting one unrouted customer into a current partial route at each iteration. Insertions based on maximum savings, minimum additional distance and time, and nearest neighbor concepts have been proposed (Ball 1995, 83). Local search techniques search a neighborhood of the current solution to find a route with a superior function value. The neighborhood of a route is the collection of all routes that

can be obtained from the current route by performing one move or transformation. Branch exchange improvement methods are easy to implement and are performed both within and between routes.

Some of the more recently published heuristic approaches are reviewed below. For a more exhaustive account, refer to Bodin et al. (1983), Desrochers et al. (1988) and Ball et al. (1995).

Baker and Schaffer (1986) use four route construction heuristics to generate initial solutions to their two test data sets. The 2-opt and 3-opt branch exchange improvement procedures are tailored to account for vehicle capacity and time window constraints. Branch exchanges are sequentially considered, first within a route and then between routes. Feasibility is maintained as routes are reconfigured. Solomon, Baker and Schaffer (1988) employ within-route branch exchange improvement procedures to significantly reduce computational effort without degrading solution quality. They look for ways to minimize the time window checks necessary to guarantee feasibility. Their results clearly show advantages in using pre-processing for 3-opt branch exchange procedures.

Solomon (1987) analyzes and compares several tour building algorithms for VRPTW. All algorithms use 2-opt refining procedures to improve the routes. His computational results revealed that a sequential time-space based insertion

algorithm outperformed the other route construction heuristics for the VRPTW. The sweep heuristic performed best on problems with many customers per vehicle. The other two insertion heuristics minimize waiting time for problems with short scheduling horizons. The time-oriented nearest neighbor heuristic also had limited success on problems with long scheduling horizons and large vehicle capacities. The savings heuristic did not perform well on any of Solomon's 100-customer benchmark problem sets. Solomon (1986) showed that the VRPTW was significantly more difficult to solve than the VRP.

Solomon's insertion heuristic is quick and effective and is used quite frequently to build initial feasible routes for other neighborhood search techniques. The main problem with this method is that the last unrouted customers tend to be widely dispersed over the geographic area yielding routes of poor quality. Potvin and Rousseau (1993) used a parallel route building philosophy coupled with a generalized regret measure to overcome the myopic weakness of sequential approaches. This procedure was tested using Solomon's 100-customer benchmark problems. They showed that the parallel approach does not work as well as a sequential approach for problems that are already clustered.

Savelsbergh (1992) studied the efficient implementation of edge-exchange improvement methods when the objective is to minimize route

duration and the departure time of a vehicle at the depot is not fixed, but has to fall within a time window. The author used a lexicographic search strategy in combination with a set of global variables to test the feasibility of 2-exchanges, backward Or- and forward Or-exchanges in constant time (also used in Van der Bruggen et al.). The set of global variables made it possible to test the feasibility of the exchange.

Koskosidis, Powell and Solomon (1992) proposed an iterative optimization-based heuristic for solving the VRPTW predicated on the Generalized Assignment Problem (GAP) Heuristic proposed by Fisher and Jaikumar (1981). There are two subproblems in their formulation; an assignment problem and the time window constrained TSP. The hard time window constraints are relaxed and added into the objective function in a Lagrangian relaxation fashion. Their algorithm achieved comparable results to the heuristics applied by Solomon (1987) and Baker and Schaffer (1986) at a significantly higher computational cost.

Balakrishnan (1993) used three simple heuristics to solve the VRPTW. His heuristics are tested against Solomon's 100-customer benchmark problems. It is important to note that Balakrishnan's algorithm did not yield better results than Solomon or Koskosidis et al. (1992) in any of the scenarios. He only

obtained better results when he softened the time window constraints. The author asserted that by controlling the amount of time window constraints that are violated, he may find a marginally infeasible solution that is attractive based on a lesser number of routes and total route distance.

Garcia, Potvin and Rousseau (1994) described a parallel Tabu search heuristic for the VRPTW. The objective used was to minimize cost with a finite number of vehicles. A special 2-opt exchange heuristic that preserves the orientation of the routes was illustrated. This was combined with Or-opt exchanges for solution improvement using Tabu search. The master processor controls the Tabu search. The "slaves" are used to explore different neighborhoods of the current solution. The master receives the best move information from each slave and is able to make several modifications simultaneously to the current solution. Their parallel algorithm improved upon the best solutions found by Solomon (1987) on the benchmark 100-customer problems but at a greater computational effort.

Rochat and Semet (1994) solved a real life routing problem involving the transportation of pet food and flour in Switzerland. The problem considered a wide variety of customers - local farmers to retailers and wholesalers - with varying demands and different resupply schedules serviced by a *heterogeneous*

fleet of vehicles from a central depot. The authors used the time-space insertion heuristic procedure proposed by Solomon (1987) to generate a quick feasible initial solution. After each insertion, a 2-optimality routine was used to improve the route. The key concept in this paper was how the authors handled accessibility. Not all customers can be serviced by all types of vehicles. The authors first solve a VRP restricted to the accessibility- constrained customers and generate partial "compact" (clustered) routes. The second phase inserted customers that are not constrained by type of vehicle.

Rochat and Semet employed a reactive tabu search strategy (to be discussed later) to improve the solution. They tested their procedure on the actual problem data. Their results were encouraging and showed how their procedure minimized the infeasibilities that were already in the routes presently used by the companies.

Recently, Carlton (1995) employed the reactive tabu search metastrategy to solve the VRPTW. Many of the initial ideas for solving the PDPTW with RTS were adopted from Carlton's work. Reactive tabu search incorporates the basic parameters and memory structures of tabu search and, in addition, uses routines allowing the algorithm to automatically adjust search parameters based on the quality of the search (Carlton 1995, 97). This search strategy permits the

search to cross infeasible regions in the state space in quest of feasible solution tours. Carlton develops a two-level open hashing structure to efficiently record the history of the solution tours. He tests his algorithm on the benchmark problem sets proposed by Solomon (1987). His algorithm returned nearly optimal tours for all problem instances in a fraction of the computational effort required by the optimal approach by Desrochers et al. (1992). Results also indicate that the reactive tabu search metastrategy regularly produces superior solutions with much less computational effort than the best known heuristics. His algorithm proved to be robust, efficient and flexible.

Potvin and Rousseau (1995) compared various iterative route improvement heuristics. Only a few heuristics are useful when time windows are present. They evaluated problems with tight, large and a mixture of tight and large time windows. One of the best known approaches for modifying solutions is the *k*-opt exchange heuristic. The complexity of this procedure is typically polynomial, but the number of iterations required to find the local optimum is exponential in the worst case. Classical *k*-opt exchange heuristics are not well adapted to problems with time windows because most exchanges do not preserve the orientation of the routes. Since customers are sequenced in accordance with some time window measure, for example, the time window's upper bound or

midpoint, reversing some portion of a route is likely to produce an infeasible solution. The authors proposed a 2-opt* exchange procedure that preserves the orientation of the routes and introduced the last customers of one route at the end of another route. Hence, the new solution is more likely to be feasible. The Or-opt heuristic was also used for finer refinements where customers are close from a spatial and temporal point of view. Their hybrid heuristic outperformed the 3-opt procedure on almost all the test data sets.

Kontoravdis and Bard (1993) present a GRASP procedure for the VRPTW including both delivery and pickup operations. The objective function is the hierarchical vehicle objective which seeks first to minimize the number of vehicles used, and secondly, to minimize the travel time. The search considers only feasible solutions and includes a vehicle reduction phase (Carlton 1995, 199).

Recently, Taillard, Badeau, Gendreau, Guertin and Potvin (1996) solved the VRPTW using Solomon's insertion heuristic to build routes and tabu search to improve the solution using a new cross exchange heuristic that preserves the orientation of the routes. They also provide an easy mechanism to swap segments of routes that are close from a spatial and/or temporal viewpoint.

2.2 Pickup and Delivery and Dial-a-Ride Problems with Time Windows

The pickup and delivery problem with time windows (PDPTW) and the dial-a-ride problem with time windows (DARPTW) are generalizations of the VRPTW. Most approaches in the literature investigate minimizing two objectives in hierarchical fashion. The algorithms attempt to first minimize the number of routes required or total fixed vehicle costs incurred. Given the minimum number of routes, the total travel time or distance is minimized. When transporting personnel, a third objective that minimizes customer inconvenience may also be considered. The hierarchical vehicle objective allows the construction of lower bounds on the number of vehicles used (Kontoravdis and Bard, 1993) which governs the notion of optimality with respect to the primary objective within the hierarchy.

The constraint set for the PDPTW is even more complex than the VRPTW. In addition to the capacity constraints on the vehicles, time window constraints for each stop and visiting constraints ensure that each stop is visited exactly once, precedence constraints guarantee that each customer is picked up before being dropped off, and coupling constraints require that the same vehicle must visit the pickup and delivery locations.

Solomon and Desrosiers (1988) survey solution techniques for solving a myriad of classes of routing problems with time windows. Solomon and Desrosiers (1988) specifically address the current exact and heuristic methods used to solve the PDPTW, to include the dial-a-ride problem (DARP). Solomon and Desrosiers (1988, 2-4) also provide a mathematical formulation for the single depot, homogeneous fleet, route length constrained PDPTW. The objective of the formulation minimizes the sum of the total travel cost, the total penalty associated with servicing customers too early or too late and the total penalty associated with routes exceeding a given duration.

2.2.1 Exact Methods

Single vehicle Dial-A-Ride (DARP) systems do not exist in practice. Nevertheless, single vehicle DARP procedures can be used as subroutines to solve large scale multivehicle DARPs. The single vehicle DARP is a constrained TSPTW with the vehicle capacity restrictions. Psaraftis (1980, 1983) developed the first exact backward and forward recursion dynamic programming algorithms respectfully to exploit this structure while minimizing customer inconvenience. To avoid imposing time window constraints at the origins and destinations, Psaraftis used a maximum position shift with respect to the ordering of known pickup and delivery times. Since both algorithms have $O(n^2 3^n)$ complexity,

where n is the number of requests, they can handle only a limited number of customers and still be computationally viable.

Sexton and Bodin (1985a, b) also minimize customer inconvenience when one-sided time windows are encountered. The algorithm applies Benders' decomposition procedure to a mixed binary nonlinear formulation that solves the routing and scheduling components individually. The scheduling component turns out to be the dual of the maximum profit network flow problem and can be solved quickly and optimally with a one-pass algorithm. This efficiency permits the overall algorithm to be computationally tractable for problems of moderate size.

Sexton and Choi (1986) also use a Benders' decomposition heuristic that involves a two-phase routing and scheduling procedure. They use a space-time heuristic procedure to generate an initial route. Benders' decomposition procedure is used to generate an optimal solution to the scheduling subproblem. The route is improved using a Lagrangian relaxation heuristic based on the coefficients of the Benders' cut. Their objective is to minimize a linear combination of total vehicle operating time and total customer penalty due to missing any of the time windows. Computational experience suggests that this algorithm is also efficient for problems of moderate size, say 17 or fewer loads.

Desrosiers, Dumas and Soumis (1986) present an optimal solution to the single vehicle DARPTW using a forward dynamic programming approach that significantly reduces the number of states generated. The objective is to find the itinerary that minimizes the total distance traveled not the total time required to service all customers. By minimizing the total distance traveled, riding time within the vehicle or customer inconvenience is not considered. Algorithm efficiency is improved by eliminating states that are incompatible with vehicle capacity, precedence and time window constraints. Eight elimination criteria are presented, discussed and illustrated in a closing simple example. Ninety-eight problems with 5 to 40 requests generated from real world data are solved.

Dumas, Desrosiers and Soumis (1991) provide an optimal algorithm that minimizes the total distance traveled. This algorithm uses a column generation scheme with a constrained shortest path as a subproblem. The constrained shortest path problem is solved using a forward dynamic programming algorithm. The algorithm works well for problems with restrictive vehicle capacity constraints. The algorithm is also touted as being capable of handling multiple depots and heterogeneous vehicles. The algorithm was tested against eight problems ranging from 19 to 55 customers, two problems being real-life scenarios in Montreal. Computational results indicate that as the total

permissible route duration increases, so does the problem difficulty. Problems with smaller time windows are easier to solve due to a lesser number of feasible solutions and to solve a multiple depot problem requires that you solve a subproblem for each depot.

2.2.2 Heuristic Methods.

The practical size that exact methods can handle is small. Heuristic algorithms are required to solve larger, real-world size problems. Given the difficulty of the Pickup and Delivery Problem and routing problems, many heuristic algorithms do not seek global optimal solutions, but rather seek to provide fast near-optimal solutions and are customized to model specific situations. In general, heuristic algorithms reported in the literature consist of two phases, route construction and route improvement. In the first phase, feasible vehicle routes are generated for a problem instance. In the second phase, the constructed routes are improved by simple procedures. However, very little research has been extended to develop heuristic methods to solve the PDPTW.

Van der Bruggen, Lenstra and Schuur (1993) employ a two-phase *local search* algorithm to determine near-optimal solutions for the single vehicle PDPTW. The construction phase starts with an infeasible tour and reduces infeasibility at each iteration by applying an objective function that penalizes the

violation of the restrictions. The infeasible solutions only violate time window constraints, not the precedence and capacity constraints. To generate an initial mildly infeasible solution that satisfies the precedence and capacity constraints, the authors sort on the midpoints of the time window intervals. The construction phase returns a feasible solution and the improvement phase continues to minimize the objective of route duration. Both phases use a variable-depth exchange procedure based on a lexicographic neighborhood search strategy and an embedded arc-exchange algorithm using seven variants of arc-exchanges. Global variables are used to track feasibility and profitability of arc-exchanges. Updates are completed in constant time in conjunction with the lexicographic search strategy. Their algorithm produced near optimal results in a reasonable amount of computation time on real-life problems, with known optimal solutions, originating from the city of Toronto.

Van der Bruggen et al. realized that their search should not be solely confined to feasible regions within the state space. They also developed an alternative algorithm based on a penalized simulated annealing algorithm that provided the power to escape local optima by accepting inferior solutions and traversing infeasible regions in the state space to find other local optima. Again,

the only constraints violated are the time window constraints. Higher quality solutions were found but at a higher computational cost.

2.3 Reactive Tabu Search (RTS)

A robust search engine is required to determine near optimal solutions in real time. This search strategy must also be able to traverse time window and capacity infeasible regions of the solution space. Once the search is in a feasible solution basin, you want to "intensify" your search to approach the local optimum solution. When the search has exhausted your search and are getting too many repeated solutions, you need to be able to alter your search trajectory to diversify your search into another region of the solution space and escape the current solution basin. Reactive tabu search (RTS) meets all these requirements and has proven to be most effective in solving this class of combinatorial optimization problems (Glover, 1996).

2.3.1 Tabu Search Algorithms from the Literature

Tabu search (Glover 1989, 1990a, 1990b) attempts to avoid becoming trapped in local optima by exploiting memory and data structures to prevent returning to a previously examined solution. The neighborhood structures imposed enable the algorithm to efficiently develop other solutions from the present solution. A subset of neighbors, the candidate list, is examined to

determine the best move available with respect to the objective function. The algorithm transitions to the best of the neighbors that is not tabu. A neighbor is tabu if it has an attribute that has appeared in a solution within a designated number of previous iterations. A critical required input parameter is the tabu length which indicates the prohibition period, the number of iterations within which the designated solution cannot be revisited. Tabu search algorithms incorporate aspiration criteria to allow a forbidden move to be accepted whenever such a move acceptance is deemed appropriate. Other procedures can be designed to encourage the algorithm to intensify or diversify the search. The algorithm terminates when a designated number of iterations or a predetermined amount of computation time is reached.

Despite the success of other algorithms, tabu search seems to be the method having the widest application for the special class of combinatorial optimization problems, vehicle routing problems. It is also the method having the greatest potential for further success and refinement (LaPorte 1992 and Glover 1996). Tabu search is the primary procedure used within this research. A detailed analysis of tabu search applications for solving the VRPTW will now be presented. A brief overview of VRP instances using tabu search will be explored since tabu search has not been applied to the PDPTW.

Stewart, Kelley and Laguna (1993) use a cluster first, route second approach for solving vehicle routing problems (VRP). Their algorithm decomposes the problem, uses both tabu search and local improvement methods as subroutines, and uses an innovative network (0, 1) integer programming structure to develop neighboring solutions. The objective considered minimizes total travel time.

Stewart et al. decompose the VRP into two subproblems: a generalized assignment problem (GAP) and a traveling salesman problem (TSP). They use tabu search to solve the GAP subproblem and allocate customers to routes. The GAP routes generated are always feasible with respect to the vehicle capacity constraints. With customer assignment complete, the vehicle is routed by solving the TSP. The best TSP solution found is used to generate the set of neighbor solutions. Neighbors are generated by solving a network flow problem; a few customers from their current routes are placed on new routes in an approximately least cost fashion (Stewart, Kelley and Laguna 1993, 11). These customers may not be reassigned to their initial routes for the designated tabu duration. This new assignment of customers to routes is used to generate new seed nodes for the GAP subproblem.

TABUROUTE is a metaheuristic developed by Gendreau, Hertz and

LaPorte (1994) that must be tailored to the shape of the particular problem being investigated. TABUROUTE does not try to maintain feasibility nor does it strive to return to feasibility. Thus, the risk of being trapped at a local optima is greatly reduced by this procedure and you do not need a feasible starting solution. Feasibility is encouraged by applying penalty terms in the objective function to minimize total travel time. The two penalty terms, used for violating vehicle capacity and route duration constraints, are dynamically updated within the algorithm's execution based upon the recent history of feasible solutions. Gendreau et al. track the best known feasible solution and the best solution regardless of feasibility. This is incorporated in the PDPTW approach described later.

Osman (1993) presents an alternative approach for solving the VRP. His route set formulation concentrates on minimizing the total cost of all routes. A solution is defined as the set $S = \{R_1, \ldots, R_p\}$ where R_j is the set of customers assigned to route *j*. Osman uses search neighborhoods that either add customers to routes or "swap" customers between routes, via the "λ-interchange" (Osman, 1993, 425). The author develops four algorithms: two local search procedures, a tabu search algorithm and a simulated annealing algorithm. All algorithms use the λ-interchange search neighborhoods. The cost of the move is computed using a 2-opt procedure in conjunction with the inter-route swaps.

The simulated annealing algorithm is a probabilistic heuristic enabling the search to escape a local optimum to traverse into other regions and cross infeasible regions of the solution space. The local search algorithms use various combinations of λ neighborhoods and one of two move acceptance strategies; choosing to accept either the first improving neighbor or the best improving neighbor. The author concludes, "tabu search schemes ... outperform the SA [simulated annealing] algorithm in solution quality and computation time. Tabu search results are also more robust than SA" (Osman 1993, 443). Additionally, Battiti (1995, 4) reports, "the often cited asymptotic convergence results of SA are unfortunately irrelevant for the application of SA to optimization. In fact, repeated local search, and even random search has better asymptotic results...approximating the asymptotic behavior of SA arbitrarily closely requires a number of transitions that for most problems is typically larger than the size of the solution space...thus, the SA algorithm is clearly unsuited for solving combinatorial optimization problems to optimality." Osman's algorithm does not use a post-processing improvement routine nor does it accept infeasible solutions during the search. Osman's results, when compared to those obtained by Gendreau, Hertz and LaPorte (1994), offer no definitive insight as to which tabu search routine is more effective.

Potvin et al. (1993) provide a tabu search heuristic for solving the

VRPTW. The authors develop two procedures that preserve the ordering of customers and feasibility of the tour. Classical *k*-opt exchange heuristics are not well adapted to problems with time windows because most exchanges do not preserve the orientation of the routes. Since customers are typically sequenced according to their time window's upper bound or midpoint, reversing some portion of a route is likely to produce an infeasible solution.

The first exchange procedure is equivalent to the "swap" λ-exchange procedure developed by Osman. This "2-opt*" procedure, replaces two arcs from the original solution with two new arcs forming two subtours. The "subtours are connected together into a single tour by linking the last customer in each subtour" (Potvin et al. 1993, 3). Hence, the new solution is likely to be feasible. The 2-opt* heuristic is useful when the two arcs to be replaced are from two different routes. It is not useful otherwise. The 2-opt* is not useful if the initial route construction heuristic produces an optimal clustering of the customers so exchanges between routes is not needed.

The other heuristic considered is the Or-opt (Or, 1976) heuristic. Or-opt considers a sequence of one, two and three adjacent customers in the solution. It moves only small sequences of customers which are "close" from a spatial and temporal point of view and inserts them at a new location while preserving the orientation of the routes. This heuristic is suitable for both intra- and inter-route

exchanges.

Potvin et al. use a hybrid heuristic in their tabu search. The 2-opt* heuristic is used during Phase 1 and considers new solutions that are very different from the current one. The Or-opt moves during Phase 2 focus on finer refinements. It also uses a route savings step within each tabu phase to try to reduce the number of vehicles used.

The route savings algorithm is used because the authors hierarchically base the quality of their search "first on the number of routes, and then, on a total route time" (Potvin et al. 1993, 6). A tabu length of five iterations is used for the prohibition period before being permitted to return to a prior solution. The difficulty with this algorithm is that the search only considers feasible moves. It is not clear how feasibility is verified or what the algorithm does if no feasible moves are available. Solomon's (1987) time-space insertion algorithm is used to generate a starting feasible solution. The authors test the heuristic against the 3-opt procedure on several random generated problems and Solomon's benchmark problems. Their hybrid heuristic outperformed the 3-opt procedure on almost all the test data sets.

Garcia, Potvin and Rousseau (1994) implement the Potvin et al. (1993) tabu search algorithm in a parallel processing environment. The master processor controls the tabu search. The "slaves" are used to simultaneously

explore different neighborhoods of the current solution. The master receives best move information from each slave and is able to make several modifications at once to the current solution.

Semet and Taillard (1993) tackle a real world VRPTW application that incorporates several practical constraints using tabu search. The problem involves 45 stores in Switzerland which place between 70 to 90 orders daily to be filled from a central depot. In addition to the delivery location, requests indicate delivery restrictions that might apply to that location. For example, each truck and trailer has both volume and weight limitations and not all stores can accept trailer deliveries.

The tabu search algorithm removes an order from a route and determines the best place to insert the order in all the other routes. The route ordering is refined using a 2-opt procedure. TS finds the best "allowed" move and transitions to the solution. Next, the algorithm computes the optimal assignment of vehicles to the new routes that have been formed.

In addition to using a random tabu list size, the authors experimented with two types of tabu list attributes. The first tabu attribute permits an order removed from a route to be returned to that route after the assigned tabu length. The second tabu attribute, developed specifically for this application, extends the tabu

length prohibition of the first attribute to any order from the same store of the route that just lost an order.

The delivery schedules obtained were superior to those previously used by the shipping facility. The authors report that the best combination of parameters is to use the second tabu attribute applied for 10,000 iterations. The paper demonstrates the ease of implementing tabu search for a practical application and tackles some interesting modeling issues.

Rochat and Semet (1994) apply tabu search to another real world VRPTW application involving the transportation of pet food and flour. The transportation costs represent a large percentage of the total cost of making and delivering the goods. Federal laws severely penalize excess route duration and excess weight in distribution of goods. This problem introduces unique modeling issues and innovative solution techniques.

The problem addressed by Rochat and Semet (1994) considers a wide variety of customers - local farmers to retailers and wholesalers - with varying demands and different resupply schedules. Deliveries to be made the following day are known in advance. Deliveries must satisfy the two time window demands for each customer to associate the times customers are available to receive goods. It appears that these windows are large. The customers are

served by a heterogeneous fleet of vehicles from a central depot. Solutions help determine what vehicles can service what area. Rochat and Semet also consider the breaks the drivers must take as required by Swiss federal law. The breaks are modeled as fictitious customers with a prescribed time window and corresponding service time. The demand associated with each dummy customer is zero. Finally, the total duration of each route must not be greater than 10 hours, 15 minutes.

The key concept in the paper is how the authors handle accessibility. Not all customers can be serviced by all types of vehicles. The authors first solve a VRP restricted to the accessibility-constrained customers and generate partial "compact" or clustered routes (Rochat and Semet, 1994, 1236). The second phase inserts customers that are not constrained by type of vehicle.

Rochat and Semet use tabu search to improve the solution. The solution space is the set of routes serving the whole set of customers while satisfying the accessibility constraints. All three constraint sets - carrying capacity, duration of routes and time windows - are relaxed. The authors use a weighted squared lateness penalty term for time window violations along with weighted penalty terms for route length and vehicle capacity violations. The objective function sums the penalty terms with the total distance traveled. The penalty terms are

dynamically updated based in the history of the search. The authors allow for infeasible solutions but only consider feasible solutions as a best solution. A move consists of removing a customer from one route and placing it in another. Once the move is made, the 2-opt routine is used to improve the routes. Because the problem is relatively small, the entire neighborhood of the solution is examined.

Rochat and Semet employ a reactive tabu search strategy. The tabu list length changes dynamically during the execution of the algorithm. If the objective function decreases by decreasing the infeasibilities, the tabu list length is decreased. This intensifies the local search. If the objective function increases, the tabu list length is likewise increased. The authors also employ an intensification strategy that examines a subset of routes that are spatially close. During this phase, new routes can be created if vehicles are available.

Rochat and Semet test their procedure on the actual problem data. Their results are encouraging and show how their procedure minimizes the infeasibilities that are already present in the routes presently used by the companies.

Taillard et al. (1996) attempt to minimize the total distance traveled from a single depot by a homogeneous fleet of vehicles to service a set of customers

with time window constraints. The time window constraints are soft which implies that the customer service at a location can commence after the time window's upper bound. The scheduling horizon constraint must still be met.

2-opt and 3-opt edge exchange heuristics are briefly discussed as methods used to improve vehicle routing solutions. Taillard et al. propose a generalized edge exchange heuristic called the "cross" exchange which is exploited within a tabu search heuristic for the VRPTW. Orientation of the routes is preserved by a cross exchange. Plus, this provides an easy mechanism to swap segments of routes that are close from a spatial and/or temporal viewpoint.

Their algorithm constructs 20 different initial solutions. Routes are built using Solomon's insertion heuristic and improved using tabu search. Results are stored in adaptive memory. As the search progresses, routes stored in adaptive memory are used to generate new starting solutions for tabu search in a manner reminiscent of the recombination or crossover operator of genetic algorithms.

To reduce computation time and intensify the search in specific regions of the state space, each initial solution is partitioned into disjoint subsets of routes, each subset being processed by a different tabu search. Improvements are made on each subproblem and then combined to form a new current solution. The decomposition changes from one iteration to the next. The general problem

solving methodology can be easily parallelized to speed up the computations. For example, many different staring solutions can be constructed from the adaptive memory and assigned to different tabu search processes running in parallel. Their algorithm is outlined below (Taillard et al. 1996, 13).

1. Construct 20 different solutions with Solomon's insertion heuristic. Apply the tabu search heuristic to each solution and store the results in adaptive memory.

2. While the stopping criterion is not met, do:
    a. Construct an initial solution from the routes found in the adaptive memory and set this solution as the current solution.

    b. For *I* iterations do:

        (1) decompose the current solution into disjoint subsets of routes.

        (2) apply tabu search on each subset of routes.
        (3) reconstruct a complete solution by merging the new routes found by the tabu search processes and set this solution as the new current solution.

    c. Store the routes of the current solution in the adaptive memory.

3. Apply a postoptimization procedure to each individual route of the best solution.

This problem-solving methodology produced many best "known" solutions on Solomon's test set and appears to be fairly robust among the different classes of problems in Solomon's test set.

2.3.2 The Reactive Tabu Search (RTS) Heuristic

The conceptual format developed by Battiti and Tecchiolli (1994) for the reactive tabu search heuristic will be utilized in this research. RTS starts with a deterministic initial tour and uses deterministic, not probabilistic, escape routines. RTS incorporates the basic parameters and memory structures of classical tabu search. RTS differs from tabu search in that it dynamically adjusts the search parameters based on the quality of the search path. Parameter tuning is performed while the algorithm runs and monitors its past behavior. The quality of the search is determined by the number of iterations occurring since the last time a particular solution has been visited. High quality solution paths seldom revisit, or "cycle", to the same solution. If a solution is repeated within a designated number of algorithm steps or "cycle length", RTS increases the tabu length search parameter in order to discourage further repetitions. If numerous, high quality solutions are being identified, RTS intensifies the search in that solution subspace to determine the best solution in that locale. When too many solutions are repeated too often, the search is assumed to be trapped in a local attractor basin. RTS provides a diversification strategy that allows the search to climb out of the basin into other regions of the solution space. The outline of the RTS algorithm follows.

1. The search moves to a neighbor solution based on the one of the three neighborhood search schemes.

2. The algorithm determines if the solution has been visited before.

    a. If the solution has been visited within the designated cycle length, the tabu length is increased by the suitable multiplicative factor.

    b. If the solution has never been visited, it is added to the solution list, and if the search conditions warrant, the tabu length is decreased by a suitable multiplicative factor.

3. If all candidate neighbors are tabu and none meet aspiration criteria, then the algorithm escapes to the neighbor with the smallest move value regardless of its tabu status, and the tabu length is decreased. This occurs when the current solution has a very small number of permissible moves.

Battiti and Tecchiolli (1994) suggest the conditions leading to a decrease in tabu length in step 2b, above. Whenever a solution is revisited in fewer steps than the designated maximum cycle length, the algorithm computes a moving average of the cycle length. If the number of iterations without changing the tabu length is greater than the moving average, then the tabu length is decreased by the multiplicative factor (Carlton 1995, 97-98). Barnes and Carlton (1995) and Carlton (1995) have successfully employed the RTS approach to solve VRPTW.

Chapter 3

Detailed Problem Description

The pickup and delivery problem with time windows (PDPTW) is a specific case of the PCRPTW. The PDPTW constructs optimal routes to satisfy transportation requests, each requiring both pickup and delivery under capacity, time window, precedence and coupling constraints. An unlimited number of homogeneous vehicles housed at one depot are available. Each transportation request requires picking up material at a predetermined location during a prescribed time window and delivering the material to a specific destination during a time window. Loading and unloading times are incurred at each vehicle stop. The PDPTW incorporates clearly defined "pairwise" precedence relationships. Each pickup location is joined to one and only one delivery location.

Other characteristics of the problem investigated in this research will be described in this chapter. Section 3.1 discusses which type of transportation request problem will be investigated. Section 3.2 overviews the various type of objectives used in vehicle routing problems. Section 3.3 lists the constraint sets that must be satisfied by the PDPTW. Section 3.4 covers the assumptions, notation and definitions used in the research.

3.1 The Advanced Request Problem

A static, or advanced request, problem is completely specified before a solution is required. This corresponds to routine information required for a system which requires customers to request service long enough in advance so that vehicles may be completely routed before departing the depot. No further requests for service are accepted after the vehicles are dispatched. The dynamic, or immediate request, problem permits requests for service to occur after the vehicles are dispatched. This research investigates the static scenario.

3.2 Objectives

There are a variety of objective functions applicable to time windows constrained routing problems. Often two or more objectives will be addressed in hierarchical fashion.

Minimize Number of Vehicles.

The purchase cost of new vehicles and/or the leasing of additional vehicles may be considerable. In the public sector, adding more vehicles to an existing fleet could necessitate fare or product price increases which in turn reduces the ridership or consumer spending. The Armed Services, moreover, must request funding from Congress to acquire additional transportation assets. The necessary approval process can be very rigorous and time consuming. Each proposal is scrutinized by subcommittees which require detailed justification for

the expenditures.

Minimize Route Length.

One measure of the operating costs of a transportation system is route length. Minimizing route length contributes to minimizing fuel and maintenance costs and driver pay. Route length is usually measured in terms of the physical distance traveled or total time required.

Maximize Customer Satisfaction.

When providing transportation service to people, it is important to consider their satisfaction with the quality of service. Customer satisfaction may be measured in a variety of ways, and different individuals may value different characteristics. For example, customers often seek to minimize total travel time, the time from when the request is made (or the desired pickup time) to the moment the individual is actually dropped off. Deviation from desired pick up and delivery times may also be important. When evacuating casualties, customer inconvenience is measured by the number of patients that must remain overnight at staging facilities.

The objective used helps determine the type of search neighborhood or neighborhoods that should be used. This research will concentrate solely on minimizing the total travel time.

3.3 Side Constraints

The PDPTW imposes constraints which must be enforced in the model. These constraint sets are:

Time Windows

Time window constraints arise whenever a customer requires the service to be performed within a certain time period. The time window depicts the customer's earliest and latest required times for service to begin and end. This research focuses on the time windows constrained routing problems for the following reasons.

Time window constraints commonly occur in real life. Major retailers desire receipt of their supplies during off-peak or after duty hours so their laborers can off load the vehicles and restock the shelves in preparation for the next business day. Time window constraints need to be modeled in this research because of its practical application to military deployment and logistics and maintenance service support.

The addition of time windows can increase the difficulty in finding feasible solutions. However, it also can enhance the basic problem structure by limiting the number of feasible solutions. Often, the more restrictive the time windows, the easier it is to solve the problem to optimality. Larger time windows tend to defeat exact approaches because of the huge number of feasible

solutions that must be evaluated and eliminated.

Time windows may be two-ended, specifying an early time and late time, or one-ended, where one time is not specified (Psaraftis, 1986). Additionally, time window constraints may be treated as either hard or soft constraints. Hard time window constraints may never be violated with respect to either the earliest or latest service time. For soft time window constraints, time windows may be violated at either the earliest or latest service time or both depending on the problem definition. Time windows violations may also be permitted if a suitable penalty cost is imposed. The larger the penalty imposed for violating a soft time windows constraint, the "harder" the time window constraint becomes. Most often, the literature treats the early service time windows as soft constraints allowing a vehicle to arrive at a customer before the early service time and wait until the service window opens. In this instance, the tour completion time may be penalized by the waiting times accrued at the corresponding customers. This research treats early time window limits as soft constraints and allows early arrival at the customers' locations.

Precedence Constraints

Precedence constraints arise whenever one activity or series of activities must occur before beginning another activity or set of activities. Precedence embodies the logical sequence of real life activities that occurs in a problem such

as a taxi cab first picking up its customer and, subsequently, delivering the customer.

The addition of precedence constraints can make it easier to build a feasible solution because of the restricted ordering that exists. This also enhances the basic problem structure by limiting the number of feasible solutions.

Same Vehicle or Coupling Constraints

Often an organization will plan a series of activities that requires workers to simultaneously support the community in a number of widely dispersed locations. Constraints need to be imposed to ensure that the same vehicle that picks up the supplies delivers those supplies to the designated customer.

Precedence and coupling constraints are "hard" constraints and will not be violated at any iteration of the search procedure. Precedence and coupling are the dominant constraints that motivated the selection of neighborhood search strategies used to solve the PDPTW (see section 4.2).

Vehicle Capacity

Vehicles are limited by the amount of supplies that they can transport, whether limited by the total weight of the items or the sheer bulk, or volume, of the supplies, or, in some applications, both weight and volume limitations are present.

Planning Horizon

The planning horizon is the time all vehicles are permitted to complete their routes. For the VRPTW, a vehicle leaves the depot either full, prepared to deliver supplies, or empty, ready to receive supplies. The vehicle capacity can be more limiting than the planning horizon in the VRPTW because the vehicle can only haul so many supplies before it must return to the depot. In the PDPTW, supplies are both received and delivered during the route. Supplies are being either added or subtracted from the total weight or volume of supplies on the supplies. With the amount of supplies transported by the vehicle fluctuating, the planning horizon can be more restrictive than the capacity restrictions.

3.4 PDPTW Assumptions, Notation and Definitions

All parameters of the PDPTW are assumed to be known with certainty. Each problem has a set P of customers requiring service, either a pickup, P^+, or a delivery, P^-, such that $|P| = n, P^+ \cup P^- = P$ and $|P^+| = |P^-| = \frac{n}{2}$. The set of n customers is indexed by subscripts i and j. Other input parameters required to define a specific instance of the problem are travel times between all pairs of locations, t_{ij}; the service time, s_i; the demand, d_i; the earliest service time, e_i; and latest service time, l_i for each of the customers. For algorithmic simplicity, if there is a nonzero s_i, it is added to all t_{ij} for customer i. Thus, nonzero s_i need not be considered explicitly by the solution algorithm after their effects are accounted

for in the t_{ij} and l_i. An important point to note is that the travel times are not symmetric for the PDPTW. The travel time, t_{ij}, records the travel time to move directly from customer *i* to customer *j*. If customer *j* is the paired successor of customer *i*, t_{ji} will be set to a large value, *M*, to make it undesirable for the algorithm to select this leg of the route. There are two other instances where the travel time between locations will be set arbitrarily large. Vehicles leave the depot empty and must travel to a supplier first, not a delivery location. The last stop a vehicle will make is to unload the remaining supplies on the vehicle. All vehicles are assumed to return to the depot empty. Travel times from the depot to a delivery location and from the supplier returning to the depot will be set to *M*. Travel times t_{ij} are based on Euclidean distance and an average rate of travel. All distances are assumed to satisfy the triangle inequality, unless otherwise indicated.

In general, a precedence ordered subset (POS) reflects a known ordering of a subset of customers to be serviced in the problem. If a customer *i* is the supplier for delivery location, customer *j*, customer *i* must be visited before customer *j*. Customer *i* is a *predecessor* of *j*, and *j* is a *successor* of *i*. However, *i* does not necessarily have to be visited immediately before *j*. This relation may be further denoted by $pred_j = i$ and $succ_i = j$. Note that the successor for customer *i* or the predecessor for customer *j* need not be unique (as explained below).

Every customer belongs to a single precedence ordered subset. The following types of precedence ordered subsets exist and will be examined during this research.

1. *pairwise*-POS. This defines the unique ordering between a supplier and its corresponding delivery used in the PDPTW. There are $n/2$ subsets, or "node pairs", defined for each PDPTW instance.

2. *serial*-POS. This defines a known ordering that must exist between three or more customers. For example, given three customers, say a, b and c, the ordering a-b-c says that customer a must be serviced before customer b who is serviced before customer c.

3. *generalized*-POS. This defines a partial ordering among three or more customers. Examples of this POS occur where a single supplier supports several delivery locations or several suppliers support a single delivery location. The single supplier must be serviced first before delivering the supplies to the customers in the first example. An algorithm minimizing distance traveled will determine the order of the deliveries serviced by that single supplier. The successor for the single supplier is a list of its corresponding delivery locations.

The *serial-* and *generalized-POSs* are discussed in chapter 5.

Each problem has a set V of available vehicles, indexed by k, such that $|V| = m$, and each vehicle has a known capacity C. All vehicle nodes will be modeled after the depot such that $\text{pred}_k = \text{succ}_k = \text{pred}_0 = \text{succ}_0 = 0$, $\text{d}_k = \text{d}_0 = 0$, $\text{s}_k = \text{s}_0 = 0$, $\text{e}_k = \text{e}_0 = 0$, and $\text{l}_k = \text{l}_0$ (section 4.1.2). Since the vehicle nodes are modeled after the depot, the travel times from customers to vehicle nodes are

$$\text{t}_{i,j} = \text{t}_{i,0} \text{ for } i \in P^-, j \in V \text{ and } j > n \tag{3.1a}$$

$$\text{t}_{i,j} = \text{t}_{0,j} \text{ for } j \in P^+, i \in V \text{ and } i > n \tag{3.1b}$$

$$t_{i,j} = M \text{ for } i \in P^+, j \in V \text{ and } j > n \tag{3.1c}$$

and

$$t_{i,j} = M \text{ for } j \in P^-, i \in V \text{ and } i > n \tag{3.1d}$$

It will be assumed that the number of vehicles required will not impose a limitation on this research. For the PDPTW, the number of vehicles can be initially set to $n/2$, the number of transportation requests or pairwise relationships that must be satisfied. N is the set of all modeled nodes which include all customers, vehicles and the depot, $|N| = n + m + 1$.

A solution is represented by a vector indicating the order in which customers are served by the unique vehicles to which they are assigned. The solution "tour" is represented by the vector T:

$$T = \{\tau_0, \tau_1, \tau_2, \ldots, \tau_{n+m-1}, \tau_{n+m}\} \tag{3.2}$$

where τ_i identifies the customer or vehicle node in the ith position of the route. By convention, the customer in position 0 and in position $n + m$ is the depot depicted as $\tau_0 = 0$ and $\tau_{n+m} = n + m$ for all tours, T. The remaining customers and vehicles may occupy any position from 1 to $n + m - 1$, inclusive. Nodes identified 1 through n are customer nodes. Nodes identified n+1 through $n + m$ are vehicle nodes. A locator array, $locx_i$, is used to record the current position of the nodes in the tour with $locx_0 = 0$ and $locx_{n+m} = n + m$ for all tours, T. The locator array is important to the search process because you can readily identify the location of a customer without having to search through the tour vector. For

example, $locx_2 = 17$ immediately tells us that customer 2 is located in position 17 instead of searching through the vector T to find out $\tau_{17} = 2$. Initially, T is set to the vector $\{0, 1, 2, \ldots, n+m\}$ before the initial tour is constructed and the search is initiated.

Since waiting is allowed, define the arrival time, A_i, departure time, D_i and waiting time, W_i, at node i as follows:

$$A_{\tau_i} = D_{\tau_{i-1}} + t_{\tau_{i-1},\tau_i} \text{ for } i \in N \tag{3.3}$$

$$D_i = \max(A_i, e_i) \text{ for } i \in P \text{ and } D_i = 0 \text{ for } i \in V \tag{3.4}$$

$$W_i = \max(0, e_i - A_i) = D_i - A_i \tag{3.5}$$

Note that $W_i > 0$ whenever a vehicle arrives at customer i prior to e_i. However, service cannot start prior to the early arrival time.

The early service time window constraints will be treated as "soft" constraints allowing the vehicle to wait if it arrives before the early arrival time. The late time window constraints are hard and must be satisfied in any feasible tour. These constraints are represented:

$$A_i \le l_i \; \forall \; i \in N. \tag{3.6}$$

If i identifies a vehicle node, equation (3.6) evaluates if the route is completed during the normal duty day or the planning horizon. Satisfying this requirement will determine if overtime is required for the drivers.

Vehicles investigated in this study have finite capacity. These constraints

must also be strictly satisfied to produce a feasible set of tours. The total load on a vehicle at any time must be less than the vehicle capacity. The vehicle capacity constraints are expressed:

$$\text{load}_k \leq C \; \forall k \in V. \tag{3.7a}$$

where

$$\text{load}_i = \text{load}_{i-1} + \text{d}_i \text{ for } i \in P \text{ and } \text{load}_k = 0 \text{ for } k \in V \tag{3.7b}$$

A penalty structure totaling the amount of time windows and load violations associated with each tour is provided below:

1. S_{TW} totals the amount of time windows violations for the tour and is computed by

$$S_{TW} = \sum_{i=1}^{n+m} \max(0, A_i - l_i) \tag{3.8}$$

where the expression A_i - $l_i > 0$ indicates either that the vehicle arrived at customer i after the late service time to service the customer or the vehicle returned late to the depot.

2. S_{ld} totals the amount of overload for a tour and is computed by

$$S_{ld} = \sum_{i=1}^{n+m} \max(0, \text{load}_i - C) \tag{3.9}$$

This information will be used for two reasons. First, the time windows and overload violations, appropriately weighted, will be added to the objective function to evaluate infeasible solutions. Additionally, the time windows and overload violations will be used in the two-level open hashing structure to further discriminate between tours (refer to section 4.3.3).

The dominant constraints that will be enforced at each iteration are the precedence and coupling constraints. All tours discovered during the search, to include infeasible tours, will be workable. In each route, the same vehicle that picks up supplies will deliver them to their delivery site. The pickup site will be visited prior to the delivery site. The precedence and coupling constraints that must be satisfied are

$$locx_i \leq locx_{succ_i} \text{ and } Veh_i = Veh_{succ_i} \; \forall i \in P^+ \tag{3.10a}$$

$$locx_{pred_j} \leq locx_j \text{ and } Veh_{pred_j} = Veh_j \; \forall j \in P^- \tag{3.10b}$$

where the notation Veh_i indicates the vehicle to which customer i is assigned.

The total travel time is the sum of all inter-customer distances in the tour. A tour's total travel time is given by $\sum_{i=0}^{n+m-1} t_{\tau_i,\tau_{i+1}}$. This research adds the time windows and capacity violations to the objective function weighted by an appropriate value because the search methodologies used during this research must traverse infeasible regions of the solution space. The penalty for time windows, $\mathrm{PEN}_{\mathrm{TW}}$, is set equal to 1 to reflect the actual time that the tour violates l_i. The penalty for capacity violations, $\mathrm{PEN}_{\mathrm{ld}}$, is set to 100 to correspond to the fine that could be levied for carrying a load in excess of the vehicle's capacity. The objective function is thus defined as

$$Z_t(\mathrm{T}) = \sum_{i=0}^{n+m-1} t_{\tau_i,\tau_{i+1}} + \mathrm{PEN}_{TW} \times \mathrm{S}_{TW} + \mathrm{PEN}_{ld} \times \mathrm{S}_{ld}. \tag{3.11}$$

Chapter 4

The PDPTW Algorithm

4.1 The Input Data and Initial Tour

The following section discusses how Solomon's benchmark data for the VRPTW was modified to generate data sets for the PDPTW. Two different route construction algorithms were also used to generate starting tours for the PDPTW.

4.1.1 The Benchmark Data Sets

The Solomon problems (1987) have long existed as the benchmark data set for VRPTW. Unfortunately, there are no clearly defined benchmark problems available to test algorithms solving the pickup and delivery problem with time windows. For this reason the Solomon problems, with appropriate modifications, were used as the test bed for the research documented herein.

The Solomon problems used in this study are the r1, c1 and rc1 data sets, having 12, 9 and 8 problem instances, respectively. (For a complete listing of the data sets modified, refer to appendix A.) The 29 data sets are written for three different customer sizes; 25-, 50- and 100-customers. The short planning horizon that characterizes these data sets further constrains the PDPTW by acting as a capacity constraint and limiting the number of customers that can be serviced by the same vehicle.

Problem set r1 has radially dispersed, randomly distributed customer

locations. Problem set c1 has clustered customer locations while problem set rc1 provides a mixture of radially dispersed and clustered customer locations. Within all three data sets customers have randomly assigned time windows. An individual problem has either 25%, 50%, 75% or 100% of its customers assigned time windows. As the percentage of customers having time windows increases, the number of feasible solutions encountered during the search decreases and the time to complete a heuristic search procedure also tends to decrease. Solomon (1987) provides a detailed discussion of the data sets generation.

4.1.2 The Input Data Structure

The initial entry in the data tells how many suppliers and deliveries must be visited. This is the number of customers, *numcust*, and will be an even number to reflect the "pairwise" precedence relationship. The first two columns of the data set provide the location of each customer consisting of an (x, y) coordinate for each customer based on the distance measure used by the analyst. This (x, y) coordinate is used to generate the time-distance matrix. Associated with each customer is the corresponding time window $[e_i, l_i]$, service time, s_i and demand, d_i. A positive demand in the third column indicates a supplier and how many supplies, by weight or volume, that will be loaded on the vehicle. A negative demand reflects a delivery being made. This indicates that the load

hauled by the vehicles will fluctuate depending on whether a pickup or delivery is being made (refer to section 3.3). The service time in column five provides how long it will take to off-load or on-load the supplies. The final two entries in the row establish the precedence relationships that must be satisfied. The first of the two entries, $pred_i$, declares which customer serves as the predecessor to node i. If $pred_i = 0$, node i does not have a predecessor node. This indicates that node i is itself the predecessor node in the paired precedence relationship. Thus, the second entry, $succ_i$, would be nonzero indicating which node is the successor to node i.

The depot node data is placed in the first row. The planning horizon is specified by the length of the depot time window and by design equals l_0 since e_0 is set to zero. The remainder of the entries for the depot node are set to zero, i.e., $d_0 = s_0 = pred_0 = succ_0 = 0$.

The format for the data follows (where $n = numcust$, the number of customers).

n							
x_0	y_0	d_0	e_0	l_0	s_0	$pred_0$	$succ_0$
x_1	y_1	d_1	e_1	l_1	s_1	$pred_1$	$succ_1$
.							
.							
.							
x_n	y_n	d_n	e_n	l_n	s_n	$pred_n$	$succ_n$

Carlton's (1995) RTS heuristic was run on Solomon's VRPTW

benchmark problems to generate the optimal or best found solution schedules used for the PDPTW problem instances. The exact VRPTW solutions of Desrosiers, Desrochers and Solomon (1992) and Kohl (1995) were used to validate the results from Carlton's RTS heuristic. Given the tour schedules, the VRPTW input data structure was modified for the PDPTW. Customers were randomly paired within each route and the predecessor and successor columns updated with the identification of the corresponding node pair. If an odd number of customers was present on a route, the customer without a pair was paired with a dummy node modeled after itself. The service time for the original node is set to zero and the dummy node gains the service time. This dummy node serves as the delivery node and successor to the original node establishing a node pair. Appendix B provides an 25-customer example illustrating how Solomon's data is modified to support the PDPTW structure.

The Solomon problems, with the modifications discussed above, were used as the best bed for this research because there are no clearly defined benchmark problems available to test algorithms solving the pickup and delivery problem with time windows. The limitation in using these modified benchmark problems is that the problem instances may not be globally representative of all possible structures and may not possess the necessary rigor to fully test the PDPTW algorithm.

4.1.3 The Optimal Solution vs Best Solution Found

Desrochers, Desrosiers and Solomon (1992) and Kohl (1995) were able to find optimal solutions for several of the VRPTW benchmark problem instances. However, many of the schedules outlining an optimal tour are unavailable. Carlton (1995) used a reactive tabu search metaheuristic to solve the VRPTW benchmark problems and found "nearly" optimal solutions in a comparatively short amount of time. Carlton's VRPTW code was used to generate the best/optimal tour that was used to provide the information to complete the modified input data for the PDPTW. Specifically, the best feasible/optimal tour found was used to determine which nodes would serve as predecessors and which nodes would be successors.

The problem instances where only the best feasible solution was used could potentially provide complications during the search process. The RTS approach attempts to find a good feasible, but not optimal tour. These competing solutions could force the search trajectory to miss locating either solution. Plus, imposing the PDPTW data structure could make infeasible an unidentified VRPTW optimal tour, since it might violate the new precedence and coupling constraints. As a result, the optimal tour might not be discovered. Finally, as discussed in the prior section, these modified data sets based on the best feasible solution found may not be as broad or diverse as desired to provide the necessary

rigor to fully test the PDPTW algorithm.

The summary results available from Desrochers, Desrosiers and Solomon (1992) and Kohl (1995) were used to identify the number of routes required for the optimal tour. Carlton's RTS algorithm was used to identify either the optimal tour or tours with an equivalent number of routes. For the 25- and 50-customer problem instances, optimal tours with an equivalent number of routes were discovered. Optimal tours with an equivalent number of routes were only found for the clustered 100-customer problems. Carlton's code discovered tours with 20, 18 and 16 routes while Kohl (1995) reported finding tours with 18, 17 and 15 routes for three of the radially dispersed 100-customer problems - r101, r102 and r105. These three problem instances were used for exploration. Infeasible tours generated by the PDPTW algorithm could possibly have an equal number of routes as the optimal tours indicated by Kohl.

4.1.4 Infeasible Initial Tour

Solomon (1987) analyzed and compared several tour building algorithms for VRPTW. All algorithms used 2-opt refining procedures to improve the routes. His computational results revealed that a sequential time-space based insertion (I1) algorithm outperformed the other route construction heuristics for the VRPTW by being the most stable in all problem environments (Solomon 1987, 256-260).

The Solomon insertion heuristic builds tours sequentially. For each new route, the unrouted customer farthest from the depot becomes the first customer assigned to the route. The remaining unrouted customers are assigned a value representing the minimum cost associated with inserting it into any feasible position in the new route (Solomon 1987, 257). The customer having the best cost is inserted into the corresponding position in the new route associated with that cost, provided the insertion is feasible. When no customer may be feasibly inserted, a new route is started. The procedure continues until all customers are feasibly placed.

Solomon's insertion heuristic is quick and effective and is used quite frequently to build initial feasible routes for other neighborhood search techniques (Carlton, 1995, 204 and Solomon, 1987). The procedure forms a feasible tour using as many vehicles as necessary without regard to actual vehicle availability. The main problem with this method is that the last unrouted customers tend to be widely dispersed over the geographic area yielding routes of poor quality.

Solomon's insertion heuristic places customers on routes based on their location and satisfaction of time windows constraints. The insertion heuristic does not account for precedence and coupling concerns. Therefore, in this work, an additional one pass sweep is conducted to ensure that precedence and coupling

constraints are satisfied. The sweep uses the locator array, $locx_i$, coupled with the Veh_i portion of the node structure to determine where the other member of the *pairwise*-POS is located and whether the pair is located on the same route, respectively. If the pair is not located on the same route, the tour is adjusted so that the pair is on the same route. If precedence is not satisfied, the successor is repositioned on the route immediately following the successor.

Conducting this one pass sweep ensures that precedence and coupling constraints are satisfied. However, the sweep does not consider satisfying time windows or capacity requirements. The tours that are created may be infeasible due to time windows violations.

4.1.5 Feasible Initial Tour with a Vehicle Reduction Phase

Previous studies have explored whether it is more advantageous to start with an initial feasible tour rather than some arbitrary starting tour. Results of these studies have indicated that it is desirable to start with an initial feasible tour (Potvin, et al., 1993, Thangiah, 1993, Kontoravdis and Bard, 1993 and Carlton, 1995). One way to create an initial feasible tour for the PDPTW would be to place each pickup location and its delivery location on the same route. If this initial tour proves to be infeasible, then there is no feasible solution satisfying all the constraints. Creating this initial feasible tour also serves as a check to ensure that the modified input data generated is accurate.

The only shortcoming using this initial feasible tour is that extra "up front" effort must be expended using an appropriate neighborhood search strategy to quickly conduct vehicle reduction. Using this initial tour, followed by a "vehicle reduction phase" is suitable as long as the problems investigated are relatively small, 50 customers or less. (Results using this initial tour are provided in appendix E.)

4.1.6 Feasible Initial Tour without the Vehicle Reduction Phase

Early investigations showed that a composite method was required to generate an initial feasible tour that did not require a vehicle reduction phase. To accomplish this, a sequential time-space *pairwise*-POS insertion algorithm was created. This routine inserts the first *pairwise*-POS on the first route. The routine then attempts to feasibly place the next *pairwise*-POS on that route. If such exists, the routine finds the feasible insert location for the *pairwise*-POS that adds the least amount to the overall tour travel time. If there is no feasible placement, a new route is created. The routine continues until all customers are scheduled. The procedure provided a feasible initial tour for all the test bed problems.

4.2 Neighborhood Search Strategies

Three search neighborhoods are hierarchically employed after the initial solution is constructed.

4.2.1 Single Precedence Ordered Subset Insertion (SPI)

The first neighborhood search scheme attempts to place a single precedence ordered subset (POS) on another route in the tour. For the PDPTW, a POS refers to a supplier and its corresponding delivery location, a *pairwise*-POS. SPI searches a route for a predecessor/supply node. Once the predecessor node is identified, the search attempts to feasibly place this node on all the other routes. To feasibly place this node on another route, the predecessor node must only satisfy time window and capacity constraints for incorporation on the alternate route. For every feasible position found for the predecessor node, the successor node is inserted after the predecessor node in ALL subsequent locations remaining on the route, from immediately following the predecessor to just prior to the depot node to finish the route. The successor node is located using the locator array, $locx_i$.. The change to the objective function is computed for each location (using the *move_delta* function as discussed in appendix C). The majority of the locations where the successor is inserted will result in infeasible tours. However, this procedure does allow infeasible tours to be

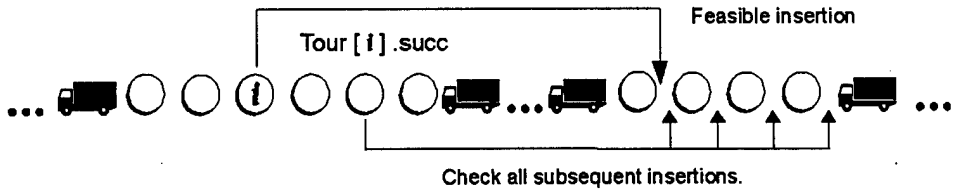

Figure 1 *An Example of Single Precedence Ordered Subset Insertion*

investigated.

An annotated description of the SPI neighborhood search is provided below.

1. Increment neighborhood counter, ++spicount;

2. Initialize neighborhood search parameters;

3. Search for a predecessor/supply node on the first (subsequent) route.

4. Attempt to feasibly insert, satisfying load and time windows constraints, the predecessor node *i* as early as possible on an alternate route.[1]

5. If a feasible insertion position is found,

    a. create a working tour.

    b. insert predecessor node on alternate route in the working tour.

    c. compute partial tour length change.

6. Insert successor node, tour[*i*].succ, in all positions subsequent to the predecessor node on the alternate route.

    a. create a second working tour based on the working tour in 5a.

    b. compute remaining tour length change based on insertion of the successor node.

    c. compute time windows and load capacity violations and adjust the tour length.

[1] If inserting the predecessor before the last node on the route, the route identifying vehicle node, check to see if the *pairwise*-POS can be inserted. Checking to insert the predecessor/supply node before the vehicle/depot node is not permitted. It is one of the inadmissible positions.

d. record results based on the (in)feasibility of the neighbor tour.

7. If at the end of the tour, stop the search and update the results; else, return to step 2 and search for the next predecessor/supply node.

8. Update tour results.

    a. If one or more feasible moves are found, move to best such move.

    b. If there is no improving pair to insert,

        (1) decrement the neighborhood counter, --spicount.

        (2) reinitialize neighborhood search parameters.

        (3) go to swap_pairs. This is the escape routine.

    c. Check to see if the neighbor move selected eliminates a route.

        (1) During the first half of the search, accept the move if the neighbor move is a feasible tour.

        (2) During the second half of the search, infeasible route reduction moves are accepted.

9. Update tabu list and tour positions. Allow no "return" moves for tabu_length iterations.

    a. Use the predecessor node position only for recording tabu moves.

    b. Allow no "repeat" moves for tabu_length iterations to avoid cycling.

10. Compute the tour hashing value.

Of all the move neighborhoods, SPI commonly provides the greatest

reduction in objective function. SPI also establishes an upper bound on the cardinality, or number, of the respective routes of the tour. The SPI search neighborhood is the only neighborhood search strategy of the three search strategies that will perform a reduction in the number of routes. However, eliminating routes will be permitted only if the new solution is feasible during the first half of the overall search process. The first half of the search attempts to find the best feasible tour that satisfies all constraints. During the second half of the search, any move eliminating routes will be accepted, even the neighbor move is infeasible.

When the modified Solomon's I1 sequential space-time insertion algorithm generates an infeasible initial tour, SPI quickly eliminates the time windows and capacity violations created by the one-pass sweep. However, SPI is an $O(n^3)$ search neighborhood and is the most expensive search mechanism used by the algorithm.

4.2.2 Swapping Pairs Between Routes

The second search neighborhood scheme swaps POSs between routes. This search neighborhood is often required to change the makeup of routes. During the search process, the SPI search routine could very possibly be "locked" out of any potential moves. This happens when there are no locations to feasibly place any of the predecessor nodes on any of the other routes. This situation is

extremely common when time windows are tight. This situation does not guarantee that the optimal solution has been found. It just means that you are locked out of using SPI to find feasible alternate neighbor tours in the solution space. By swapping *pairwise*-POSs between routes, you overcome the barrier and alter the search trajectory to find new areas of the feasible solution space.

This neighborhood search is also used for diversification when the search trajectory is confined to a limited portion of the solution space with no clear periodicity. Too many solutions are being revisited during a prescribed number of iterations. This is a chaotic attractor basin. A predetermined series of swapping *pairwise*-POS between routes determined by the size of the problem is executed. This drastically alters the complexion of the route. The tabu search parameters are reinitialized and the search continues from this new solution in a different region of the solution space.

This search neighborhood swaps a POS pair for another POS pair on another route. No attempt is made to find the best place to insert the successors after the predecessors are swapped. This type of fine tuning is relegated to the

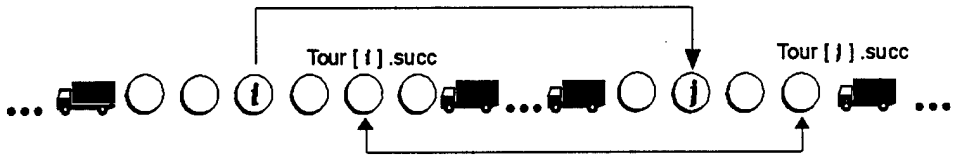

Figure 2 *The Swap Pairs Neighborhood Search*

neighborhood described in section 4.2.3. An annotated description of the swap pairs neighborhood search procedure is provided below.

1. Increment neighborhood counter, ++swapcount.
2. Initialize neighborhood search parameters;
3. Search for a predecessor/supply node on the first (subsequent) route.
4. Search for a predecessor node on an alternate route later in the tour.
5. If swapping the predecessor nodes does not violate tabu length restrictions,
    a. create a working tour.
    b. swap the *pairwise*-POSs between the routes.
    c. compute move evaluation.
    d. compute time windows and load capacity violations and adjust the tour length.
    e. record results based on the (in)feasibility of the neighbor tour.
6. If at the end of the tour, stop the search and update the results; else, return to step 2 and search for the next predecessor/supply node.
7. Update tour results.
    a. If one or more feasible moves are found, move to the best such move.
    b. If there are no improving pairs to swap, use the escape criteria and select the best move found.
8. Update tabu list and tour positions. Allow no "return" moves for tabu_length iterations.

a. Use the predecessor node position only for recording tabu moves.

b. Allow no "repeat" moves for tabu_length iterations to avoid cycling.

9. Compute the tour hashing value.

This search often selects a new tour that is infeasible. This is allowable because the purpose of this search is to alter the makeup of the routes and the search is not constrained to feasible tours. Time window or capacity infeasible neighbor tours are critical to the search process and could provide a viable tour having significantly lower objective values than the optimum. If the new tour is infeasible, the next search neighborhood will attempt to make minor modifications within the routes to see if the nodes on the respective routes can be feasibly arranged or arranged so as to further minimize the time windows and capacity violations.

4.2.3 Within Route Insertion (WRI)

The third search neighborhood is used to fine tune travel time within routes by moving individual locations earlier or later in their respective routes. The other two search neighborhoods can generate infeasible tours that violate time windows or capacity. This search neighborhood will attempt to reorder the customers to further minimize or negate the time windows violations and to make

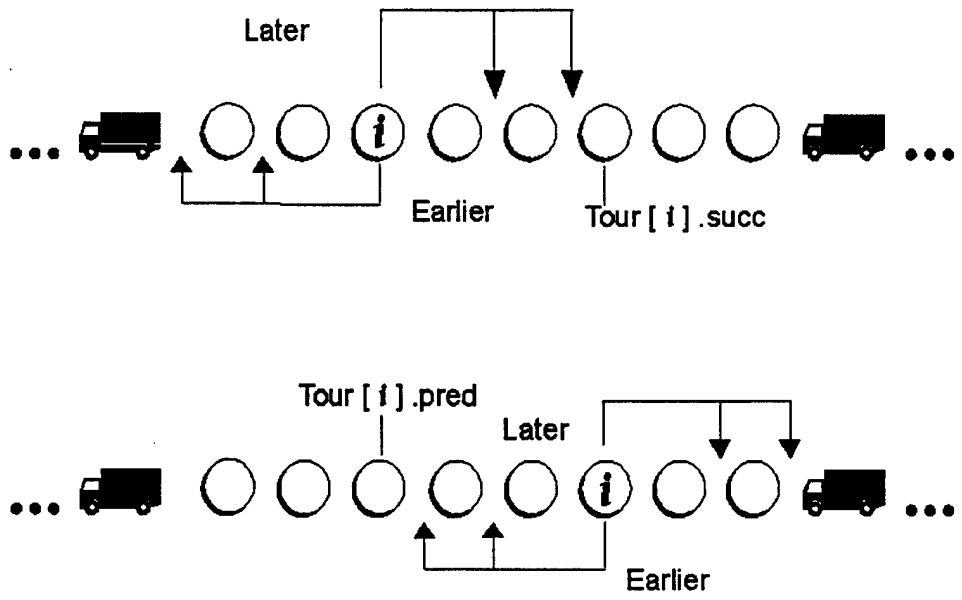

Figure 3 *The WRI Neighborhood Search*

minor improvements in the objective function. This search neighborhood is especially required when large time windows are prevalent. Numerous feasible solutions are available when large time windows are present. Altering the customer arrangement within the routes will explore other possible orderings to determine the best possible order for the routes.

WRI limits the search to within routes instead of across routes. Moving individual locations across routes requires too much additional computational effort to track precedence and coupling violations.

The WRI search is further restricted by strong time window *infeasibility* and precedence considerations. Strong time window infeasibility occurs when

one customer cannot be feasibly serviced before or after another customer. For example, customer i is said to be strongly time window infeasible with respect to customer j if $e_i + t_{ij} > l_i$. Precedence constraints are enforced ensuring that all solutions are workable and make sense; suppliers occur before their corresponding delivery and both customers are placed on the same route for all *pairwise*-POSs in the problem. The following annotated description looks for improvement within the respective routes only.

1. Increment neighborhood counter, ++wricount;
2. Initialize neighborhood search parameters;
3. Conduct LATER within route insertions.
    a. create a working tour.
    b. attempt to insert a customer node later in the route if it does not violate precedence or strong time window feasibility.
    c. compute tour length change using *move_delta*.
    d. compute time windows and load capacity violations and adjust the tour length.
    e. record results based on the viability of the neighbor tour.
    f. if at the end of the route, the vehicle node, move to subsequent routes and proceed with 3.a. above.
    g. if at the end of the tour, continue with step 4.
4. Conduct EARLIER within route insertions.

a. create a working tour.

b. attempt to insert a customer node earlier in the route if it does not violate precedence or strong time window feasibility.

c. compute tour length change using *move_delta*.

d. compute time windows and load capacity violations and adjust the tour length.

e. record results based on the viability of the neighbor tour.

f. if at the end of the route, the vehicle node, move to subsequent routes and proceed with 4.a. above.

g. if at the end of the tour, continue with step 5.

5. If there are no escape moves (there is a strong possibility given tight time windows),

   a. decrement the neighborhood counter, --wricount.

   b. go to the SPI neighborhood search routine.

6. If one or more feasible moves are found, move to the best such move.

7. If all moves are tabu, but an escape move exists, move to it and decrease the tabu length.

8. Update tabu list and tour positions. Allow no return and repeat moves for tabu_length iterations.

9. Compute the tour hashing value.

4.2.4 Limiting the Search

It is very simple to form a feasible solution for the PDPTW as shown earlier if, indeed, a feasible solution exists (see section 4.1.5). However, the set

of feasible solutions for the PDPTW is not necessarily a connected space with respect to a given neighborhood definition. The more constraints that are present in the problem, the more likely that feasible solutions will be isolated from one another. If the feasible solution space is not connected, an algorithm only investigating feasible solutions can find an optimal tour only if the initial tour resides in a subspace that contains one or more optimal tours. Van der Bruggen, Lenstra and Schuur (1993) show that there is no guarantee that any reasonable neighborhood structure ensures the connectivity of feasible solutions for the single vehicle PDPTW. The example they constructed demonstrates disconnected feasibility regions in the general case for any reasonable neighborhood (Van der Bruggen et al. 1993, 308-309). Their research also revealed that as the time windows narrow, there is an increased likelihood of having a number of feasible solutions isolated from one another with regard to any practical move structure.

It is very simple to form precedence viable solutions for the PDPTW, as explained earlier. Precedence viable solutions are "workable" in that the required ordering of picking up the supplies prior to making the delivery is satisfied plus the correct supplies will be on the vehicle that makes the delivery. The SPI search neighborhood is capable of eliminating routes in the tour. The swap pairs neighborhood search is critical to altering the makeup of the routes. Finally,

WRI provides the requisite fine tuning by reordering customers within routes. The neighborhood search routines do not consider tours that violate precedence and coupling constraints. While not proven, empirical results obtained during this research supports the conjecture that the set of precedence viable solutions is a connected subset based on the three neighborhood searches.

4.2.5 Time Windows Reduction

A common characteristic of optimal algorithms in solving vehicle routing problems is their heavy reliance on time window relationships to amply trim the dynamic programming (DP) state-space and to restrict the number of feasible solutions so that the DP approaches are computationally tractable. The larger the time window width, the greater the number of feasible solutions that must be explored. The effectiveness of any DP based algorithm is limited because the size of the resulting state space grows exponentially as the width of the time windows increases. Serious consideration has been given to develop methods to reduce time window width.

Desrochers, Desrosiers and Solomon (1992) outline four techniques that are applied sequentially and iteratively to reduce the width of time windows. Kontoravdis and Bard (1992) employ a similar approach to reduce time windows based on the travel time from the depot. These procedures were developed for vehicle routing problems that did not include precedence relationships, i.e., no

predefined ordering existing between the nodes. The time windows reduction process is drastically streamlined for PDPTW because of the precedence relationships that exist.

Dumas, Desrosiers and Soumis (1986) develop the obvious method for reducing time windows for PDPTW. "A preliminary step in the determination of the admissible arcs is the shrinking of the time windows associated with the pickup and delivery nodes. This is done by reducing the upper bounds of the time windows so that for $i = 1, \ldots, n$, the partial paths $n + i$ [successor]$\rightarrow 2n + 1$ [depot] and i [predecessor]$\rightarrow n + i$ [successor]$\rightarrow 2n + 1$ [depot] are admissible for all values $T_i \in [e_i, l_i]$ and $T_{n+i} \in [e_{n+i}, l_{n+i}]$" (Dumas, Desrosiers and Soumis, 1986, 10). The upper and lower bounds for the time windows are successively defined by

$$l_{n+i} := \min \{l_{n+i}, l_{2n+1} - t_{i.2n+1} \tag{4.2}$$

and

$$l_i := \min \{l_i, l_{n+i} - t_{i,n+i} \quad \text{for } i = 1, \ldots, \text{n}. \tag{4.3}$$

$$e_i := \max \{e_i, e_0 + t_{0,i} \tag{4.4}$$

and

$$e_{n+i} := \max \{e_{n+i}, e_i + t_{i,n+i}\} \tag{4.5}$$

Equation (4.2) states that the upper bound on the time window for delivery/successor nodes could possibly be further constrained by the time it takes to make that final delivery and return to the depot. Similarly, equation (4.4) indicates that the lower bound for supply/predecessor nodes could further

be limited by the time it takes to travel from the depot to the supply point. Equations (4.3) and (4.5) reveal that the late departure time for the supplier and the corresponding early arrival time for the delivery are constrained only by the *pairwise*-POS ordering - that the supply node i must occur before its corresponding delivery node $n+i$ is serviced where n refers to the number of "ordered" pairs defined in the PDPTW.

4.2.6 Inadmissible Arcs

Numerous researchers have developed methods to eliminate arcs from the network structure, thereby eliminating the need to consider arcs that lead to obvious infeasibilities. The predefined ordering that exists for PDPTW coupled with the time windows and load constraints eliminates the following inadmissible arcs.

a. Arcs from the depot to delivery locations and from suppliers immediately returning to the depot are eliminated because of violating precedence.

b. Arcs from the delivery node to its corresponding supplier are eliminated because of violating precedence.

c. Arc (i, j) is eliminated if $e_i + s_i + t_{ij} > l_j$, strong time windows infeasibility.

d. Arcs (i, j), (j, i), $(i, succ_j)$, $(succ_i, succ_j)$ and $(succ_j, succi_)$ are eliminated if vehicle capacity is violated, i.e., where $d_i + d_j$ >C, where nodes i and j are supply nodes.

4.3 Two-Level Open Hashing Structure

A key issue to the effective implementation of RTS is the accurate identification of previously visited solutions. The two-level hashing scheme proposed by Horowitz, Sahni and Freed, 1993, and implemented by Carlton, 1995, will be used in this research. The two-level hashing structure has proven to be effective in minimizing collisions, where two nonidentical solutions are incorrectly determined to be duplicate solutions. In addition, the two-level hashing scheme efficiently stores solutions. It would not be efficient to store the solution tour vector for every solution visited during the search, and it definitely would not be computationally effective to compare the current tour with every previously visited tour position-by-position to determine if the current tour has been visited.

The two-level hashing scheme is implemented as follows.

1. Once a tour is accepted as the incumbent tour, compute

    a. the hashing function, f(T) = Z(T)mod k, based on the objective function value, Z(T), where k is a large prime number. This is the first-level hashing.

    b. the tour hashing value, which transforms the tour solution vector into an unsigned integer. thv(T) = $\sum_{i=0}^{n} \Psi(\tau_i)(\tau_{i+1})$, where τ_i is the index of the customer assigned to tour position i. $\Psi(i)$ is a randomly generated integer in the range (1, 131,072) for each modeled node.

2. Compare the associated penalty for time windows, P_{tw}(T), and thv(T)

for the incumbent tour to the values of tours linked to hash table element f(T).

3. If both values $P_{TW}(T)$ and thv(T) match the stored values, then the tour is being revisited.

4. If the tour is revisited, compute the cycle length and change the tabu length, if required.

5. If the tour is visited for the first time, add the tour to the hashing table and reduce the tabu length if required (Carlton 1995, 102-103).

For a complete overview on implementing the two-level open hashing structure proposed by Horowitz, Sahni and Freed (1993), refer to Carlton (1995, 99-103) and Carlton and Barnes (1996, 237-239) article on hashing.

4.4 The Tabu Criteria, Tabu Length and Data Structures

Short-term memory functions are used to determine whether a solution with a characteristic attribute has been visited before. If the algorithm discerns that a candidate solution possesses attributes of a recently visited solution within a specified tabu length, the move is disallowed and the next candidate move is entertained. The selection of the tabu attribute, the associated data structure and the tabu length are vital design features which contribute to the efficiency and success of the search. Whenever a move yields an objective function value lower than previously discovered, the aspiration criterion is invoked and the tabu status of such a move is overridden.

The algorithm uses an $(N + 1) \times (N + 1)$ array, **tabu_list**[i, j], to record

and enforce the tabu status. The row index identifies the customer, $\tau_i = 0, ..., n+m$ where τ_i is the identification number of the customer assigned to tour position i. The column index records the tour position. The array elements store the iteration number *after* which the customer can return to that position in the tour. Customer τ_i might move from its current position by being chosen as an allowable candidate move or it might move indirectly corresponding to repositioning of other customers in the tour. The tabu restriction must consider both attributes to define a move's prohibition period. Failure to account for both conditions may cause "indirect cycling" among two or more customers. For a further discussion and example of indirect cycling, refer to Carlton (1995, 117-118). Thus, when the search determines that customer τ_i is to move from position i to position j, the value k + *tabu_length,* where k is the current iteration count, is stored in two locations. Setting **tabu_list**[τ_i,j] makes all "repeat" moves tabu for *tabu_length* iterations. A repeat move is any subsequent move of node τ_i into position j, the position into which node τ_i is moving. The value k + *tabu_length* is also stored in location **tabu_list**[τ_i, i] making all "return" moves tabu for the prohibition period. A return move would permit node τ_i to reenter the position is just vacated. A modification is made for the WRI search neighborhood when the move is to an adjacent later location. Since a return move is based on node index τ_{i+1}, the value k + *tabu_length* is stored in array

location **tabu_list**$[\tau_{i+1}, i+1]$.

The algorithm performs a simple check to determine if a candidate move satisfies both tabu conditions. If the current iteration number $k \leq$ **tabu_list**$[\tau_i, j]$, then the proposed move of customer τ_i from position i to position j is not allowed unless it leads to a globally superior objective function value.

Two of the search neighborhoods move *pairwise*-POSs between routes. It is only necessary to update the **tabu_list**[i, j] array for the predecessor nodes in the *pairwise*-POS. Possible insertion positions of the successor node using SPI are based solely on finding a feasible insertion point for the predecessor. Similarly, the swap pairs neighborhood search finds predecessor nodes on differing routes and evaluates the possibility of swapping the pairs between the routes. The purpose behind the swap pairs neighborhood search routine is to alter the makeup of the route through diversification or overcome a wall in the landscape when the SPI routine is locked out of any moves. Setting **tabu_list**$[\tau_i, j]$ = **tabu_list**$[\tau_i, i]$ = **tabu_list**$[\tau_j, i]$ = **tabu_list**$[\tau_j, j]$ = k + *tabu_length*, where i and j are the positions of the predecessor nodes, accomplishes the purpose.

The final tabu criteria to consider is the tabu length. Computational results indicate that the tabu length should be based on the size of the problem. Thus, the initial tabu length is set to

$$tabu\_length = \max(30, \text{number of } pairwise\text{-POSs}) \quad (4.6)$$

This initial value gives consistently good results in conjunction with the aforementioned tabu attributes.

4.5 The Algorithms

Six algorithms were explored during this research; BUILD, BUILDc, NEW, NEWc, INIT and INITc. The algorithms differ in their initial tours and the methods used to modify the candidate list during the SPI search neighborhood. The BUILD, BUILDc and INITc algorithms were used for initial investigations and served as a foundation for future algorithms.

BUILD and BUILDc algorithms use the most basic feasible tour - placing each *pairwise*-POS, the supplier along with its corresponding delivery location, on the same unique route - as the initial tour (refer to section 4.1.5). This is followed by a "vehicle reduction phase" to eliminate the majority of excess vehicles initially used. The first few iterations, equivalent to the number of *pairwise*-POSs in the problem, use the SPI neighborhood search scheme to reduce the number of routes.

The INIT and INITc algorithms use Solomon's insertion heuristic along with an additional one pass sweep to correct precedence and coupling violations to generate an infeasible initial tour (refer to section 4.1.4). The majority of the vehicle reduction has been completed in creating the initial tour. Therefore, extra

up-front effort performing successive SPI iterations to reduce the number of routes is not necessary. Some SPI iterations are required to clean up the tour and remove the time window infeasibilities created by the one pass sweep.

The NEW and NEWc algorithms expend more effort to generate a route reduced feasible initial tour. The NEW and NEWc algorithms use a sequential time-space *pairwise*-POS insertion algorithm to generate a feasible starting tour with a significantly reduced number of routes in the starting tour when compared to the BUILD algorithms starting feasible tour (refer to section 4.1.6).

The SPI search neighborhood (section 4.2.3) used in the BUILD, INIT and NEW algorithms examines all possible *pairwise*-POS insertions in the neighborhood and accepts the most improving non-tabu move. The BUILDc, INITc and NEWc algorithms accelerate the SPI search process by accepting the first improving non-tabu move discovered. The remainder of the algorithms proceed in similar fashion. A annotated description of the algorithms will be presented later in this section.

The *multineighborhood strategic search methodology* specifies the scheme for alternating between search neighborhoods to generate a quality search path. All six algorithms use the same hierarchical form of multineighborhood strategic search methodology to direct the search. This hierarchical form is based

on average time window length (*atwl*). The *atwl* is computed after time windows reduction. *Atwl* provides an indication as to the number of potential feasible solutions that may exist in the solution space.

$$atwl = \frac{\sum_{i=1}^{n} l_i - e_i}{numcust} \tag{4.7}$$

In this study, using the test bed problems, it was observed that if *atwl* is greater than 25% of the route duration length, numerous feasible solutions exist and minor adjustments within routes are required to fine tune, or intensify, the search and possibly discover better tours. After performing one iteration of the SPI neighborhood search, perform *numcust*/10 successive iterations using the within route insertion (see section 4.2.3) neighborhood search.

If *atwl* is less than 25% of the route duration length, more strong time windows infeasibilities exist limiting the number of candidate moves for within route insertions. After performing one iteration of the SPI neighborhood search, perform *numcust*/25 successive iterations of WRI. The "tightness" of the problem's time windows will also determine if the WRI and/or SPI neighborhood searches get "locked out" of having any allowable neighbor moves. If this happens while using WRI, the search escapes to the SPI neighborhood search routine. If the SPI scheme does not have a neighbor move to transition to, the search escapes to the swap pairs neighborhood. If the swap pairs neighborhood encounters this phenomena, the routine invokes the escape criteria and transitions

to the best move available. This establishes the hierarchy for moving between search neighborhoods.

Local minimum points are attractors of the search trajectory generated by deterministic local search. One of the problems that must be solved is how to continue the search beyond the local minimum and how to avoid the confinement of the search trajectory. Confinements occur because the search trajectory tends to be biased towards tours with low cost function values, and, therefore, also towards the abandoned local minimum. The fact that the search trajectory remains close to the minimum for some iterations is clearly a desired effect in the hypothesis that better points are preferentially located in the neighborhood of good suboptimal points rather than among randomly extracted points.

Simple confinements can be cycles, an endless repetition of a sequence of solutions during the search. Confinements can be more complex trajectories with no clear periodicity that restricts the search to a limited portion of the solution space. This is a chaotic attractor basin. If too many solutions are revisited during a prescribed number of iterations, the search is confined to an attractor basin (Battiti 1995, 10-11).

All six algorithms provide an escape strategy if the search gets trapped in a chaotic attractor basin. If too many solutions are revisited too often during a prescribed number of iterations, escape by performing successive iterations of the

swap pairs search neighborhood to drastically alter the makeup of the tour and move the search into a different region of the solution space. Empirical studies indicated that the number of successive iterations had to be based on the size of the problem and the number of routes in the tour. Setting the number of iterations too low did not sufficiently alter the makeup of the route to escape the attraction of the basin. Use too many iterations and the search never recovers from the amount of infeasibilities generated. The number of successive swap pairs iterations performed is

$$\min(\text{number of vehicles used}, \textit{numcust}/10). \tag{4.8}$$

If the search gets caught again in an attractor basin, restart from the best tour found until that point and reinitialize the search parameters. This will alter the search trajectory and move the search into a different region of the solution space. The other place to restart the search is at a different starting point. If the search gets caught a fourth time in an attractor basin, it may not be beneficial to restart at the best tour again. Stop the current search and revisit the initial tour. Deterministically perform several successive iterations of SPI on the initial tour to create a new starting tour. Reset the search parameters and begin the search from this new starting point.

The algorithms also use the restart mechanism when few, if any, repeat solutions have been identified during the first half of the search. Empirical

results for the algorithms used in this research indicate that, in general, the best solutions are often found early in the search process. The results also reveal that if the optimal tour was missed by the initial search trajectory, soon after restarting the search at the best tour, the optimal tour was found For example, the optimal tours were found for the 25-customer problems nrc101, using INIT, and nr105, using NEWc, after restarting the search at the best tour midway through the search.

The number of iterations used in the search can affect the search process for the problem instances where the optimal tour is not found until later in the search. The smaller the number of iterations required for the search, the increased likelihood that halfway through the search, the search will restart at the best tour. If the optimal tour had not been found prior to restarting the search, the optimal tour may be found just several iterations after restarting the search with the different search trajectory. For example, two problem instances, nc102 and nc104, were solved using the BUILD algorithm for different numbers of iterations. The results are provided in Table 1. The optimal tour was found for problem nc102 44 iterations after restarting the search at the best tour for the first two iteration counts. For iteration counts of 300 and higher, the optimal tour was found before restarting the search. The optimal tour was also found slightly earlier in the search than when 200 iterations were used. For problem nc104, the

Problem	# of Allowed Iterations	Iterations to Optimal Tour
nc102	100	94
	200	144
	300-750	125
nc104	100	54
	200	104
	400	204
	500-750	211

Table 1 - Effect of iterations on the search - Restart at $\frac{1}{2}$ Iterations Allowed

optimal tour was discovered just 4 iterations after restarting the search until the iteration count reached 500 iterations. These results indicate that some preliminary runs using smaller iteration counts could prove beneficial.

There exist a couple of concerns with the BUILD routine. The first centers on the tabu structure. The problem is best explained by use of the following example. The symbol o is used to represent the vehicle nodes. The first □ used in the example refers to the depot node in position zero. The numbers refer to customer nodes and also refers to the customer's initial position in the tour. Given the initial tour where each single precedence ordered subset (POS) is placed on individual route:

□ 1 2 □ 4 5 □ 7 8 □ 10 11 □ 13 14 □ ...

Performing one iteration of SPI yields the following tour

□ 1 2 □ 4 5 □ □ 10 11 □ 13 14 7 8 □ ...

SPI moves node pair (7, 8) to route 5 eliminating the third route. The tabu list for SPI only records the position of the predecessor.

$$\text{tabu\_list}[7,7] = \text{tabu\_list}[7,13] = 1 + \textit{tabu\_length}.$$

Recall that the first tabu listing restricts return moves. The second aids in prohibiting cycling. Performing the second iteration of SPI (during the initial vehicle reduction phase) yields

□ 1 2 □ □ □ 4 5 10 11 □ 13 14 7 8 □ ...

Node pair (4, 5) is moved to route 4 eliminating the second route.

$$\text{tabu\_list}[4,4] = \text{tabu\_list}[4,6] = 2 + \textit{tabu\_length}.$$

The third iteration of SPI moves node pair (1, 2) to route 5 eliminating the original route 1.

□ □ □ □ 4 5 10 11 □ 13 14 1 2 7 8 □ ...

$\text{tabu\_list}[1,1] = \text{tabu\_list}[1,11] = 3 + \textit{tabu\_length}.$

The optimal solution is reachable by moving node pair (7, 8) to route 3.

□ □ □ □ 4 5 10 7 8 11 □ 13 14 1 2 □ ...

This move will not be permitted because $\text{tabu\_list}[7,7] = 1 + \textit{tabu\_length}$ The search will move off and perform numerous iterations before you overcome the "absolute" tabu_length requirement and are permitted to return to a prior solution. By the time the tabu restriction is lifted, the existing tour will be

drastically skewed and far away from the optimal tour in the solution space.

It is possible to "recover" and return to find the optimal solution later on in the search. Above is just a missed opportunity. The additional precedence and coupling constraints coupled with the neighborhood search strategies often encounter the optimal, or best, solution early in the search. The remainder of the search looks for precedence viable tours that are infeasible by violating time windows and capacity constraints. However, the structure of some problems will not return you to the basin containing the optimal solution. Suppose that

□ □ □ □ 4 5 10 7 8 11 □ 13 14 1 2 □ ...

is the best tour. The optimal solution can be obtained by forcing a pure restart after some predetermined number of iterations. Reinitializing , the SPI neighborhood search will yield the optimal solution.

Another approach would be to "clean up the tour" to solve the absolute tabu restriction criteria. Again, given the initial tour where a single POS is placed on a route:

□ 1 2 □ 4 5 □ 7 8 □ 10 11 □ 13 14 □ ...

Performing one iteration of SPI moves node pair (7, 8) to route 5 eliminating the third route.

□ 1 2 □ 4 5 □ □ 10 11 □ 13 14 7 8 □ ...

tabu_list[7, 7] = tabu_list[7, 13] = 1 + *tabu_length*. If we "cleaned up" this tour by

moving the vehicle node (which is used to identify the number of the route in the structure node representation for the tour) to the end of the tour with the other extraneous vehicle nodes, the tour would become

$$\square\ 1\ 2\ \square\ 4\ 5\ \square\ 10\ 11\ \square\ 13\ 14\ 7\ 8\ \square\ \square\ \ldots$$

The tabu_list would have to be adjusted to read

$$\text{tabu\_list}[7,7] = \text{tabu\_list}[1,12] = 1 + \textit{tabu\_length}$$

to account for the change in the geometry. The locator array would have to be updated to reflect this change. The second iteration of SPI now yields

$$\square\ 1\ 2\ \square\ \square\ 4\ 5\ 10\ 11\ \square\ 13\ 14\ 7\ 8\ \square\ \square\ \ldots$$

by moving node pair (4, 5) to route 4 eliminating the second route. After the clean-up we have

$$\square\ 1\ 2\ \square\ 4\ 5\ 10\ 11\ \square\ 13\ 14\ 7\ 8\ \square\ \square\ \square\ \ldots$$

The tabu_list would have to be further adjusted to reflect

$$\text{tabu\_list}[7,6] = \text{tabu\_list}[7,11] = 1 + \textit{tabu\_length} \text{ and}$$

$$\text{tabu\_list}[4,4] = 2 + \textit{tabu\_length}.$$

Notice that this latter tabu listing is really redundant because node 4 starts and finishes in position 4. Hence, this restriction could be effectively eliminated. Node pair (4, 5) is free to move anywhere in the tour.

The third iteration of SPI moves node pair (1, 2) to route 5 eliminating the original route 1. The tour becomes

□ 4 5 10 11 □ 13 14 1 2 7 8 □ □ □ □ ...

after the vehicle node associated with route 1 is moved to the end of the tour. Because route 1 was eliminated, all current tabu listings will have to be decremented by one in the second, or *j*, column. This records the initial positions for the (precedence) nodes as if the vehicle nodes for routes eliminated were not even included in the tour. Thus,

$$\text{tabu\_list}[7,5] = \text{tabu\_list}[7,10] = 1 + \textit{tabu\_length} \text{ and}$$

$$\text{tabu\_list}[1,1] = \text{tabu\_list}[1,8] = 3 + \textit{tabu\_length}.$$

The optimal solution is now reachable by moving node pair (7, 8) to route 3.

□ 4 5 10 7 8 11 □ 13 14 1 2 □ □ □ □ ...

This move to position 4 is permissible since there is no tabu restriction.

This example problem reveals two critical concerns. The first comes from using the most basic feasible tour - placing each *pairwise*-POS, the supplier along with its corresponding delivery location, on the same unique route - as the initial tour. This scenario increases the likelihood of encountering the tabu restriction difficulty. Coupled with this is the need to attempt to eliminate the "excess" nodes created by the BUILD algorithm. The example shows how the presence of these extra vehicle nodes could affect the tabu restrictions. Plus, carrying around the extra nodes during the search process bogs down the computation time because of the unnecessary comparisons being performed.

This causes the search process to become computationally intractable when the problem instances get large. This problem is overcome by using the INIT and NEW algorithms.

The number of vehicles, *num_vehs,* modeled in a problem for all six algorithms is initially set to the number of *pairwise*-POSs in the problem. Thus, the total number of nodes, *nnodes*, is

$$nnodes = numcust + num\_vehs + 1 \tag{4.9}$$

One additional vehicle is added to this total to provide a way of stopping the search at the end of the tour. When you get to the last position in the tour, say position *i*, the last vehicle node in the tour will have the same identification number as its position in the tour, i.e.,

$$\tau_i = Veh_i = i \tag{4.10}$$

One of the benefits of using the INIT and NEW algorithms is that the excess vehicle nodes could be trimmed after creating the initial tour with very little effort. The total number of nodes is trimmed by setting the number of vehicles equal to the number of routes in the initial tour created by the INIT or NEW algorithms, plus one additional vehicle node to mark the end of the tour.

The algorithm is described below.

0. Initialize: structures, vectors and search parameters.
1. Input problem instance.

a. Number of iterations = *niters*.

b. Compute the time/distance matrix.

c. Model vehicle nodes after the depot node.

2. Compute the time windows reduction.

3. Construct the initial tour.
   - the most basic feasible tour for the BUILD algorithms.
   - the infeasible tour for the INIT algorithms.
   - the reduced route feasible tour for the NEW algorithms.

   a. Generate the initial schedule.

   b. Compute the initial tour cost = travel time + waiting time.
   c. Compute the initial hashing values - f(T) and thv(T).

4. While $k \leq niters$ do

   a. Look for incumbent tour in the hashing structure.

      (1) If found, update the iteration when incumbent tour was found, increase the tabu length, if applicable.

      (2) If not found, add tour to the hashing structure, decrease the tabu length, if applicable.

   b Vehicle Reduction Phase (for BUILD only).

      (1) Perform *n*POS successive iterations of SPI where *n* is the number of *pairwise*-POSs.

      (2) Store ending solution for future reference as the initial tour.

   c. Conduct a hierarchical neighborhood search scheme.

      (1) Perform one (1) iteration of SPI followed by fine

tuning using WRI. The number of iterations of WRI is determined by the average time window length, *atwl*.

(2) If SPI causes reduction in the number of routes, and if $k \leq \frac{niters}{2}$

(a) permit move if the neighbor route is feasible.

(b) disallow move if the neighbor route is infeasible; go to swap pairs.

(c) else if $k > \frac{niters}{2}$, allow reduction in routes.

(3) If WRI gets "locked out" of performing any move because of strong time windows infeasibilities and precedence constraints, go to SPI.

(4) If SPI gets "locked out" of performing any transition, go to swap pairs.

(5) Move to the non-tabu neighbor according to the appropriate decision criteria. If all tours are tabu, move to the neighbor with the smallest move value and reduce the tabu length.

d. If you get caught in a chaotic attractor basin, diversify by performing successive swap pairs iterations equivalent to min(number of vehicles used, *numcust*/10), equation (4.8).

e. If you get caught again in a chaotic attractor basin, restart at the best tour found by the search up to that point. Reinitialize search parameters.

f. Update the search parameters.

(1) Incumbent tour schedule.

(2) Incumbent tour hashing value.

(3) Retain the best feasible solution found and the tour

with the smallest tour cost regardless of feasibility.

h. $k = k + 1$.

5. Output results.

The algorithm is coded in ANSI standard C programming language. The time to complete the search starts with the computation of time windows reduction and ends prior to recording the results.

4.6 Computational Results

This section presents the computational results of the three competitive reactive tabu search algorithms - INIT, NEW and NEWc - in solving the modified Solomon's (1987) problem set. Appendix A describes this set of problems in detail. This section compares the computational results of three RTS algorithms to the optimal VRPTW approaches of Desrochers, Desrosiers and Solomon (1992) and Kohl (1995) and the VRPTW reactive tabu search approach of Carlton (1995).

All three algorithms are coded in C and the runs were conducted on an IBM RISC 6000 workstation. The code was compiled with the standard C compiler using the -O3 optimization flag. The RTS algorithm was run using the following parameter settings.

1. $PEN_{TW} = 1.0$. This is the multiplicative factor used to weight the total amount of infeasibility with respect to time windows. Setting PEN_{TW} to 1 results in the penalty term equaling the total lateness in the tour.

2. PEN_{ld} = 100.0. This is the multiplicative factor used to weight the total amount of infeasiblity with respect to load capacity violations.

3. tabu_length = max(30, number of pairwise-POSs).

4. The tabu_length increase factor is set to 1.2. This is the multiplicative factor by which the tabu_length is increased if a solution is revisited within the designated cycle length.

5. The tabu_length decrease factor is set to 0.9. This is the multiplicative factor by which the tabu_length is decreased, if search conditions warrant it.

6. Cycle length = 50. If a solution is revisited within 50 iterations, the tabu_length is increased by the multiplicative factor of 1.2.

7. Hash table array dimension = 1009. This is large prime number comparable to most of the objective values anticipated.

The exact algorithm by Desrochers, Desrosiers and Solomon (1992, 350) is coded in FORTRAN and executed on a SUN SPARC 1 workstation. Kohl used a HP 9000-735 computer and coded the algorithm in PASCAL. The HP 9000-735 is over eight times faster than the SUN SPARC 1 workstation used by Desrochers, Desrosiers and Solomon (Kohl and Madsen 1995, 30). The VRPTW RTS metaheuristic by Carlton (1995) is coded in C and compiled on an IBM RISC 6000 workstation. Despite the differences in computing platforms and codes, the NEW tables in this section demonstrate the significant decrease in computation time the NEW tabu search algorithm achieves over the other algorithms. This decrease occurs despite the added data structure required for the

modified PDPTW data sets and the additional information that must be processed resulting from this added structure. This improvement is achieved at a very small decrease in solution quality. Additionally, infeasible tours with reduced objective function values are discovered for some of the problem instances.

4.6.1 25-customer problems

Tables 2, 4 and 6 record the results for the nc1, nr1 and nrc1 25-customer problem instances. Column one identifies the problem instance. Columns two and three display the minimum travel time and the number of vehicles required to achieve the travel time. Columns four and five reflect the number of iterations and the seconds of computation time required to locate the RTS solution, respectively. Column six shows the deviation of the RTS travel time from the optimum expressed as percentage. Columns seven through nine show the optimal results obtained by Carlton's (1995, 210-211) VRPTW code.

The best infeasible tours found are presented in tables 3, 5 and 7. Column one identifies the problem instance. Columns two and three display the reduced total travel time and the time windows penalty associated with the infeasible tour, respectively. Column four shows the percent improvement over the optimum solution. 750 iterations were used for the runs except where noted.

Problem Instance	$Z_t(T)$	Vehs Used	Iter to Best	Time to Best	% Dev	Optim $Z_t(T)$	Iter to Best	Comp Time
nc101	2,441.3	3	6	0.18	0	2,441.3	2	0.07
nc102	2,440.3	3	650	13.1	0	2,440.3	82	0.67
nc103	2,440.3	3	11	0.55	0	2,440.3	671	5.93
nc104	2,436.9	3	126	4.37	0	2,436.9	1,038	9.92
nc105	2,441.3	3	29	0.38	0	2,441.3	3	0.07
nc106	2,441.3	3	6	0.21	0	2,441.3	2	0.06
nc107	2,441.3	3	13	0.49	0	2,441.3	3	0.07
nc108	2,441.3	3	16	0.64	0	2,441.3	5	0.09
nc109	2,441.3	3	20	0.84	0	2,441.3	6	0.1
nr101	867.1	8	13	1.03	0	867.1	54	0.4
nr102	806.8	7	6	0.33	1.22	797.1	1,039	8.23
nr103	704.6	6	633	14.16	0	704.6	963	8.2
nr104	666.9	4	17	0.9	0	666.9	195	1.81
nr105	780.5	6	22	0.52	0	780.5	56	0.43
nr106	715.4	5	10	0.4	0	715.4	962	7.51
nr107	674.3	4	79	1.26	0	674.3	194	1.69
nr108	647.3	4	10	0.54	0	647.3	67	0.66
nr109	691.3	5	8	0.35	0	691.3	722	5.88
nr110	694.1	5	14	0.44	0	694.1	919	8.27
nr111	678.8	4	45	0.82	0	678.8	181	1.59
nr112	648.2	4	18	0.72	0.81	643	63	0.73
nrc101	711.1	4	7	0.12	0	711.1	1,077	7.26
nrc102	601.8	3	24	0.33	0	601.8	1,994	16.53
nrc103	582.8	3	3	0.19	0	582.8	890	7.38
nrc104	556.6	3	29	0.75	0	556.6	659	5.55
nrc105	661.3	4	375	3.77	0	661.3	180	1.31
nrc106	595.5	3	50	0.64	0	595.5	39	0.31
nrc107	548.3	3	7	0.19	0	548.3	1,412	12.37
nrc108	544.5	3	30	0.7	0	544.5	63	0.67
		Avg =	78	1.69	0.07	Avg =	466	3.92

Table 2 - INIT results, 25-customers

On average, 78 iterations were required to find the best solution in a time

of 1.69 seconds for the INIT algorithm. This represents a savings of 83.3% for number of iterations and 57% in time to find the best solution when compared to the optimal VRPTW solutions of Carlton. The overall difference in solution quality from the optima was 0.07%. 27 of 29 optimal solutions were matched. Table 3 displays two solutions that violated the time windows constraints in total amounts less than 20 time units and possessed lower objective values.

Problem Instance	**$Z_t(T)$**	**$Penalty_{tw}$**	**Vehs Used**	**% Dev**
nrc101	683.4	10.2	4	-3.9
nrc105	603.4	11.3	3	-8.76

Table 3 - INIT infeasible results, 25-customers

Problem Instance	$Z_t(T)$	Vehs Used	Iter to Best	Time to Best	% Dev	Optim $Z_t(T)$	Iter to Best	Comp Time
nc101	2,441.3	3	0	0.01	0	2,441.3	2	0.07
nc102	2,440.3	3	4	0.12	0	2,440.3	82	0.67
nc103	2,440.3	3	192	4.38	0	2,440.3	671	5.93
nc104	2,436.9	3	21	0.69	0	2,436.9	1,038	9.92
nc105	2,441.3	3	0	0.01	0	2,441.3	3	0.07
nc106	2,441.3	3	0	0.01	0	2,441.3	2	0.06
nc107	2,441.3	3	0	0.02	0	2,441.3	3	0.07
nc108	2,441.3	3	0	0.06	0	2,441.3	5	0.09
nc109	2,441.3	3	0	0.1	0	2,441.3	6	0.1
nr101	867.1	8	5	0.16	0	867.1	54	0.4
nr102	806.8	7	13	0.23	1.22	797.1	1,039	8.23
nr103	704.6	5	21	0.37	0	704.6	963	8.2
nr104	666.9	4	10	0.33	0	666.9	195	1.81
nr105	780.5	6	23	0.36	0	780.5	56	0.43

nr106	715.4	5	8	0.14	0	715.4	962	7.51
nr107	674.3	4	0	0.03	0	674.3	194	1.69
nr108	647.3	4	56	1.44	0	647.3	67	0.66
nr109	691.3	5	236	2.28	0	691.3	722	5.88
nr110	694.1	5	21	0.31	0	694.1	919	8.27
nr111	678.8	4	18	0.28	0	678.8	181	1.59
nr112	643	4	110	2.86	0	643	63	0.73
nrc101	711.1	4	371	1.02	0	711.1	1,077	7.26
nrc102	601.8	3	0	0.02	0	601.8	1,994	16.53
nrc103	582.8	3	4	0.1	0	582.8	890	7.38
nrc104	556.6	3	0	0.05	0	556.6	659	5.55
nrc105	661.3	4	7	0.1	0	661.3	180	1.31
nrc106	595.5	3	17	0.14	0	595.5	39	0.31
nrc107	548.3	3	10	0.1	0	548.3	1,412	12.37
nrc108	544.5	3	16	0.3	0	544.5	63	0.67
		Avg =	40	0.55	0.04	Avg =	466	3.92

Table 4 - NEW results, 25-customers

On average, 40 iterations were required to find the best solution in a time of 0.55 seconds for the NEW algorithm. This represents a savings of 91.4% for number of iterations and 85.9% in time to find the best solution when compared to the optimal VRPTW solutions of Carlton. The overall difference in solution quality from the optima was 0.04%. 28 of 29 optimal solutions were matched. Table 5 displays four solutions that violated the time windows constraints in total amounts less than 20 time units and possessed lower objective values.

Problem	$Z_t(T)$	$Penalty_{tw}$	Vehs Used	% Dev
nr102	765.5	2.4	6	-3.96
nr103	696	15.8	5	-2.45
nr110	688	0.1	5	-0.88
nrc101	708.3	0.7	4	-0.39
nrc105	596.5	17.3	3	-9.8

Table 5 - NEW infeasible results, 25-customers

Problem Instance	$Z_t(T)$	Vehs Used	Iter to Best	Time to Best	% Dev	Optim $Z_t(T)$	Iter to Best	Comp Time
nc101	2,441.3	3	0	0.01	0	2,441.3	2	0.07
nc102	2,440.3	3	4	0.08	0	2,440.3	82	0.67
nc103	2,445.4	3	3	0.1	0.21	2,440.3	671	5.93
nc104	2,436.9	3	21	0.61	0	2,436.9	1,038	9.92
nc105	2,441.3	3	0	0.01	0	2,441.3	3	0.07
nc106	2,441.3	3	0	0.01	0	2,441.3	2	0.06
nc107	2,441.3	3	0	0.02	0	2,441.3	3	0.07
nc108	2,441.3	3	0	0.05	0	2,441.3	5	0.09
nc109	2,441.3	3	0	0.1	0	2,441.3	6	0.1
nr101	867.1	8	5	0.16	0	867.1	54	0.4
nr102	820.6	7	12	0.18	2.95	797.1	1,039	8.23
nr103	704.6	5	30	0.27	0	704.6	963	8.2
nr104	666.9	4	10	0.28	0	666.9	195	1.81
nr105	780.5	6	390	1.66	0	780.5	56	0.43
nr106	715.4	5	8	0.1	0	715.4	962	7.51
nr107	674.3	4	0	0.03	0	674.3	194	1.69
nr108	647.3	4	196	2.77	0	647.3	67	0.66
nr109	691.3	5	25	0.29	0	691.3	722	5.88
nr110	694.1	5	645	6.36	0	694.1	919	8.27
nr111	678.8	4	20	0.17	0	678.8	181	1.59
nr112	643	4	233	3.32	0	643	63	0.73
nrc101	711.1	4	160	0.41	0	711.1	1,077	7.26
nrc102	601.8	3	0	0.01	0	601.8	1,994	16.53

nrc103	582.8	3	4	0.11	0	582.8	890	7.38
nrc104	556.6	3	0	0.05	0	556.6	659	5.55
nrc105	661.3	4	60	0.4	0	661.3	180	1.31
nrc106	595.5	3	17	0.07	0	595.5	39	0.31
nrc107	548.3	3	10	0.1	0	548.3	1,412	12.37
nrc108	544.5	3	16	0.2	0	544.5	63	0.67
		Avg =	64	0.62	0.11	Avg =	466	3.92

Table 6 - NEWc results, 25-customers

On average, 64 iterations were required to find the best solution in a time of 0.62 seconds for the NEWc algorithm. This represents a savings of 86.3% for number of iterations and 84.2% in time to find the best solution when compared to the optimal VRPTW solutions of Carlton. The overall difference in solution quality from the optima was 0.11%. 27 of 29 optimal solutions were matched. Table 7 displays three solutions that violated the time windows constraints in total amounts less than 20 time units and possessed lower objective values.

Problem	$Z_t(T)$	$Penalty_{tw}$	Vehs Used	% Dev
nr103	696	15.8	5	-2.45
nr110	688	0.1	5	-0.88
nrc105	603.4	11.3	3	-8.76

Table 7- NEWc infeasible results, 25-customers

4.6.2 50-customer problems

Tables 8-19 record the overall results for problems r1, c1 and rc1 for the 50-customer problem instances for the INIT, NEW and NEWc algorithms.

Desrochers, Desrosiers and Solomon (1992, 351-352) claimed they were able to solve 14 of the 29 problems to optimality. Carlton (1995, 212-213) used their results for comparison with his RTS heuristic solutions. Kohl (1995, 179-188) claimed that he was able to solve 27 of the 29 problems to optimality. However, the optimal schedules for these solutions were not provided. For comparison purposes, Carlton's VRPTW code was used to identify the optimal or a best tour along with the corresponding schedule. This schedule was used to create the PDPTW data. Results recorded in these tables follow the same format used for the 25-customer problems except where noted. First, column three lists the number of vehicles required by the PDPTW algorithms for the best tour found. If this differs from the number of vehicles required for the optimal or best tour result, the number of vehicles required is listed in the parentheses in column three. Tables 8, 12 and 16 display the results for the 15 50-customer problem for the INIT, NEW and NEWc algorithms, respectively, where the optimal solution schedules are known in detail. Tables 9, 13 and 17 display the results for the remaining 14 50-customer problems where the PDPTW input structure was modeled against an inferior solution found by Carlton's VRPTW code. Columns seven through nine in these tables show the results obtained by Carlton's RTS heuristic VRPTW code. Tables 10, 14 and 18 display the results for the 50-customer problems against the optimal results obtained by Kohl (1995,

210-211). Kohl was not able to identify the optimal solution for two problems, r108 and r112. Finally, the best infeasible tours found that reduce the objective value are presented in tables 11, 15 and 19. 750 iterations were used for the runs.

Problem Instance	$Z_t(T)$	Vehs Used(#)	Iter to Best	Time to Best	% Dev	$Z_t(T)$ by Kohl	Iter to Best	Comp Time
nc101	4,862.4	5	34	2.36	0	4,862.4	24	2.07
nc102	4,861.4	5	307	307	0	4,861.4	38	14.89
nc103	4,904.2	5	157	25.33	0.88	4,861.4	34	26.45
nc104	4,864.7	5	355	83.07	0.19	4,855.6	53	153.56
nc105	4,868.2	5	553	33.9	0.12	4,862.4	17	1.5
nc106	4,862.4	5	28	3.11	0	4,862.4	19	1.34
nc107	4,862.4	5	44	7.28	0	4,862.4	26	5.43
nc108	4,862.4	5	359	46.04	0	4,862.4	23	3.53
nc109	4,871.5	5	327	94.39	0.19	4,862.4	18	3.29
nr102	1,425.5	10(11)	41	17.96	1.17	1,409	15	3.5
nr106	1,308.4	9	434	32.2	1.19	1,293	30	14.9
nr107	1,288.2	8	654	79.28	6.37	1,211.1	255	324.5
nr110	1,309.4	8	594	71.19	9.39	1,197	38	27.13
nrc105	1,455	9	119	9.39	7.36	1,355.3	90	70.23
nrc108	1,098.1	6	114	11.59	0	1,098.1	82	598.9
		Avg =	274	54.94	1.79	Avg =	50	83.41

Table 8 - 50-customer INIT results where optimal schedules are known

On average, 274 iterations were required to find the best solution in a time of 54.94 seconds for the INIT algorithm. This represents a savings of 34% in time to find the best solution when compared to the exact VRPTW solutions of Kohl. The overall difference in solution quality from the optima was 1.79%. Six of 15 optimal solutions were matched and a very competitive solution using one less vehicle was identified for problem nr102.

For the remaining problems, 155 iterations, on average, were required to find the best solution in a time of 17.65 seconds for the INIT algorithm. This represents a savings of 92.8% in time to find the best solution when compared to the RTS heuristic VRPTW solutions of Carlton. The overall difference in solution quality from the best solutions was 1.91%. Five of 14 best solutions were matched. A solution using fewer vehicles was found for mixed problem nrc101.

Problem Instance	$Z_t(T)$	Vehs Used(#)	Iter to Best	Time to Best	% Dev	$Z_{best}(T)$ Carlton	Iter to Best	Comp Time
nr101	1,580.6	12	275	55.45	2.26	1,545.5	97	3
nr103	1,292.9	9	48	8.83	1.18	1,277.9	736	56.83
nr104	1,156.8	7	113	18.14	0.93	1,146.1	1,238	98.78
nr105	1,439.4	10(9)	56	7.75	2.7	1,401.7	6,854	381.31
nr108	1,124.1	6	65	13.23	0	1,124.1	4,748	388.45
nr109	1,290.9	8	189	18.15	0	1,290.9	7,046	454.7
nr111	1,212.1	7	227	34.22	0	1,212.1	6,278	461.85
nr112	1,205.6	8(6)	118	21.41	5.73	1,140.4	164	14.8
nrc101	1,466.6	9(10)	27	2.73	0.05	1,467.4	3,973	189.65
nrc102	1,342.3	8	38	4.63	0.25	1,338.9	7,143	362.43
nrc103	1,279.4	7(6)	422	30.83	5.46	1,213.2	6,811	408.39
nrc104	1,047.5	5	32	3.95	0	1,047.5	3,346	251.16
nrc106	1,228	6	263	8.66	0	1,228	3,443	189.15
nrc107	1,237.7	7(6)	307	19.14	8.15	1,144.4	2,434	170.71
		Avg =	155	17.65	1.91	Avg =	3,879	245.09

Table 9 - INIT results for remaining 50-customer problems

Problem Instance	$Z_t(T)$	Vehs Used(#)	Iter to Best	Time to Best	% Dev	$Z_t(T)$ by Kohl	Iter to Best	Comp Time
nc101	4,862.4	5	34	2.36	0	4,862.4	24	2.07
nc102	4,861.4	5	307	307	0	4,861.4	38	14.89
nc103	4,904.2	5	157	25.33	0.88	4,861.4	34	26.45
nc104	4,864.7	5	355	83.07	0.19	4,855.6	53	153.56
nc105	4,868.2	5	553	33.9	0.12	4,862.4	17	1.5
nc106	4,862.4	5	28	3.11	0	4,862.4	19	1.34
nc107	4,862.4	5	44	7.28	0	4,862.4	26	5.43
nc108	4,862.4	5	359	46.04	0	4,862.4	23	3.53
nc109	4,871.5	5	327	94.39	0.19	4,862.4	18	3.29
nr101	1,580.6	12	275	55.45	2.37	1,544	19	5.86
nr102	1,425.5	10(11)	41	17.96	1.17	1,409	15	3.5
nr103	1,292.9	9	48	8.83	1.57	1,272.9	157	136.09
nr104	1,156.8	7(6)	113	18.14	2.79	1,125.4	676	3,105.7
nr105	1,439.4	10(9)	56	7.75	2.87	1,399.3	56	32.86
nr106	1,308.4	9(8)	434	32.2	1.19	1,293	30	14.9
nr107	1,288.2	8(7)	654	79.28	6.37	1,211.1	255	324.5
nr109	1,290.9	8	189	18.15	0.32	1,286.8	568	523.91
nr110	1,309.4	8(7)	594	71.19	9.39	1,197	38	27.13
nr111	1,212.1	7	227	34.22	0.41	1,207.2	4,574	5,054.3
nrc101	1,466.6	9(8)	27	2.73	1.57	1,444	47	26.44
nrc102	1,342.3	8(7)	38	4.63	1.5	1,322.5	1,315	2,209.5
nrc103	1,279.4	7(6)	422	30.83	5.66	1,210.9	53	33.74
nrc104	1,047.5	5	32	3.95	0.16	1,045.8	173	137.61
nrc105	1,455	9(8)	119	9.39	7.36	1,355.3	90	70.23
nrc106	1,228	6	263	8.66	0.39	1,223.2	121	135.16
nrc107	1,237.7	7(6)	307	19.14	8.31	1,142.7	375	623.09
nrc108	1,098.1	6	114	11.59	0	1,098.1	82	598.9
		Avg =	226	38.39	2.03	Avg =	329	491.69

Table 10 - INIT results, 50-customers

On average, 226 iterations were required to find the best solution in a time of 38.4 seconds for the INIT algorithm. This represents a savings of 31% for

number of iterations and 92% in time to find the best solution when compared to the exact VRPTW solutions of Kohl. The overall difference in solution quality from the optima was 2.03%. Six of 27 optimal solutions were matched. Table 11 displays three solutions that violated the time windows constraints in total amounts less than 20 time units and possessed lower objective values.

Problem	**$Z_t(T)$**	**$Penalty_{tw}$**	**Vehs Used**	**% Dev**
nrc101	1,435.1	12	9	-0.62
nrc103	1,261.6	8.6	7	4.19
nrc105	1,388.4	6.7	8	2.44

Table 11 - Infeasible tours using INIT

Problem Instance	$Z_t(T)$	Vehs Used	Iter to Best	Time to Best	% Dev	$Z_t(T)$ by Kohl	Iter to Best	Comp Time
nc101	4,862.4	5	0	0.04	0	4,862.4	24	2.07
nc102	4,861.4	5	11	1.02	0	4,861.4	38	14.89
nc103	4,861.4	5	7	1.61	0	4,861.4	34	26.45
nc104	4,855.6	5	50	11.87	0	4,855.6	53	153.56
nc105	4,862.4	5	0	0.07	0	4,862.4	17	1.5
nc106	4,862.4	5	0	0.06	0	4,862.4	19	1.34
nc107	4,862.4	5	0	0.1	0	4,862.4	26	5.43
nc108	4,862.4	5	0	0.23	0	4,862.4	23	3.53
nc109	4,862.4	5	8	2.71	0	4,862.4	18	3.29
nr102	1,409	11	263	25.04	0	1,409	15	3.5
nr106	1,296.3	8	93	3.37	0.26	1,293	30	14.9
nr107	1,211.1	7	138	10.86	0	1,211.1	255	324.5
nr110	1,197	7	392	31.79	0	1,197	38	27.13
nrc105	1,355.3	8	538	15.97	0	1,355.3	90	70.23
nrc108	1,098.1	6	62	4.86	0	1,098.1	82	598.9
		Avg =	104	7.31	0.02	Avg =	50	83.41

Table 12 - 50-customer NEW results where optimal schedules are known

On average, 104 iterations were required to find the best solution in a time of 7.31 seconds for the NEW algorithm. This represents a savings of 91.2% in time to find the best solution when compared to the exact VRPTW solutions of Kohl. The overall difference in solution quality from the optima was 0.02%. Fourteen of 15 optimal solutions were matched.

For the remaining problems, 173 iterations were required, on average, to find the best solution in a time of 11.83 seconds for the NEW algorithm. This represents a savings of 93.6% in time to find the best solution when compared to

Problem Instance	$Z_t(T)$	Vehs Used	Iter to Best	Time to Best	% Dev	$Z_{best}(T)$ Carlton	Iter to Best	Comp Time
nr101	1,545.7	12	5	0.72	0.01	1,545.5	97	3
nr103	1,277.9	9	385	30.2	0	1,277.9	736	56.83
nr104	1,156.9	7	77	9.2	0.94	1,146.1	1,238	98.78
nr105	1,401.7	9	101	2.95	0	1,401.7	6,854	381.31
nr108	1,124.1	6	247	37.14	0	1,124.1	4,748	388.45
nr109	1,290.9	8	362	18.3	0	1,290.9	7,046	454.7
nr111	1,212.1	7	181	14.49	0	1,212.1	6,278	461.85
nr112	1,140.4	6	140	17.88	0	1,140.4	164	14.8
nrc101	1,474.1	10	20	0.6	0.46	1,467.4	3,973	189.65
nrc102	1,342.7	8	314	16.41	0.28	1,338.9	7,143	362.43
nrc103	1,213.2	6	31	1.54	0	1,213.2	6,811	408.39
nrc104	1,047.5	5	14	1.19	0	1,047.5	3,346	251.16
nrc106	1,228	6	185	3.39	0	1,228	3,443	189.15
nrc107	1,173	6	362	11.58	2.5	1,144.4	2,434	170.71
		Avg =	173	11.83	0.3	Avg =	3,879	245.09

Table 13 - NEW results for remaining 50-customer problems

the RTS heuristic VRPTW solutions of Carlton. The overall difference in

solution quality from the best solutions was 0.3%. Nine of 14 best solutions were matched.

Problem Instance	$Z_t(T)$	Vehs Used(#)	Iter to Best	Time to Best	% Dev	$Z_t(T)$ by Kohl	Iter to Best	Comp Time
nc101	4,862.4	5	0	0.04	0	4,862.4	24	2.07
nc102	4,861.4	5	11	1.02	0	4,861.4	38	14.89
nc103	4,861.4	5	7	1.61	0	4,861.4	34	26.45
nc104	4,855.6	5	50	11.87	0	4,855.6	53	153.56
nc105	4,862.4	5	0	0.07	0	4,862.4	17	1.5
nc106	4,862.4	5	0	0.06	0	4,862.4	19	1.34
nc107	4,862.4	5	0	0.1	0	4,862.4	26	5.43
nc108	4,862.4	5	0	0.23	0	4,862.4	23	3.53
nc109	4,862.4	5	8	2.71	0	4,862.4	18	3.29
nr101	1,545.7	12	5	0.72	0.11	1,544	19	5.86
nr102	1,409	11	263	25.04	0	1,409	15	3.5
nr103	1,277.9	9	385	30.2	0.39	1,272.9	157	136.09
nr104	1,156.9	7(6)	77	9.2	2.8	1,125.4	676	3,105.7
nr105	1,401.7	9	101	2.95	0.17	1,399.3	56	32.86
nr106	1,296.3	8	93	3.37	0.26	1,293	30	14.9
nr107	1,211.1	7	138	10.86	0	1,211.1	255	324.5
nr109	1,290.9	8	362	18.3	0.32	1,286.8	568	523.91
nr110	1,197	7	392	31.79	0	1,197	38	27.13
nr111	1,212.1	7	181	14.49	0.41	1,207.2	4,574	5,054.3
nrc101	1,474.1	10(8)	20	0.6	2.08	1,444	47	26.44
nrc102	1,342.7	8(7)	314	16.41	1.53	1,322.5	1,315	2,209.5
nrc103	1,213.2	6	31	1.54	0.19	1,210.9	53	33.74
nrc104	1,047.5	5	14	1.19	0.16	1,045.8	173	137.61
nrc105	1,355.3	8	538	15.97	0	1,355.3	90	70.23
nrc106	1,228	6	185	3.39	0.39	1,223.2	121	135.16
nrc107	1,173	6	362	11.58	2.65	1,142.7	375	623.09
nrc108	1,098.1	6	62	4.86	0	1,098.1	82	598.9
		Avg =	133	8.15	0.42	Avg =	329	491.69

Table 14 - NEW results, 50-customers

On average, 133 iterations were required to find the best solution in a time of 8.2 seconds for the NEW algorithm. This represents a savings of 59.5% for number of iterations and 98.3% in time to find the best solution when compared to the exact VRPTW solutions of Kohl. The overall difference in solution quality from the optima was 0.42%. 14 of 27 optimal solutions were matched. Table 15 displays three solutions that violated the time windows constraints in total amounts less than 20 time units and possessed lower objective values.

Problem	**$Z_t(T)$**	**$Penalty_{tw}$**	**Vehs Used**	**% Dev**
nrc101	1,428.3	12	9	-1.09
nrc103	1,244.5	0.4	7	-6.9
nrc108	1,028.4	9.4	5	-6.35

Table 15 - Infeasible tours using NEW

On average, 126 iterations were required to find the best solution in a time of 6.74 seconds for the NEWc algorithm results in Table 16. This represents a savings of 92% in time to find the best solution when compared to the exact VRPTW solutions of Kohl. The overall difference in solution quality from the optima was 0.07%. Thirteen of 15 optimal solutions were matched and problem nr102 yielded a solution with fewer vehicles.

Problem Instance	$Z_t(T)$	Vehs Used(#)	Iter to Best	Time to Best	% Dev	$Z_t(T)$ by Kohl	Iter to Best	Comp Time
nc101	4,862.4	5	0	0.05	0	4,862.4	24	2.07
nc102	4,861.4	5	11	1.02	0	4,861.4	38	14.89

nc103	4,861.4	5	7	1.74	0	4,861.4	34	26.45
nc104	4,855.6	5	226	34.48	0	4,855.6	53	153.56
nc105	4,862.4	5	0	0.06	0	4,862.4	17	1.5
nc106	4,862.4	5	0	0.05	0	4,862.4	19	1.34
nc107	4,862.4	5	0	0.11	0	4,862.4	26	5.43
nc108	4,862.4	5	0	0.22	0	4,862.4	23	3.53
nc109	4,862.4	5	8	2.67	0	4,862.4	18	3.29
nr102	1,417.9	10(11)	190	10.89	0.63	1,409	15	3.5
nr106	1,293	8	107	2.37	0	1,293	30	14.9
nr107	1,211.1	7	414	16.25	0	1,211.1	255	324.5
nr110	1,197	7	599	20.5	0	1,197	38	27.13
nrc105	1,361.7	8	156	3.41	0.47	1,355.3	90	70.23
nrc108	1,098.1	6	175	7.24	0	1,098.1	82	598.9
		Avg =	126	6.74	0.07	Avg =	50	83.41

Table 16 - 50-customer NEWc results where optimal schedules are known

Problem Instance	$Z_t(T)$	Vehs Used(#)	Iter to Best	Time to Best	% Dev	$Z_{best}(T)$ Carlton	Iter to Best	Comp Time
nr101	1,545.7	12	5	0.75	0.01	1,545.5	97	3
nr103	1,294.8	9	94	2.79	1.33	1,277.9	736	56.83
nr104	1,147.4	7	327	20.68	0.11	1,146.1	1,238	98.78
nr105	1,414.7	9	49	0.98	0.93	1,401.7	6,854	381.31
nr108	1,124.1	6	475	26.15	0	1,124.1	4,748	388.45
nr109	1,290.9	8	376	8.75	0	1,290.9	7,046	454.7
nr111	1,212.1	7	193	6.74	0	1,212.1	6,278	461.85
nr112	1,140.4	6	252	14.42	0	1,140.4	164	14.8
nrc101*	1,466.6	9(10)	692	11.36	-0.05	1,467.4	3,973	189.65
nrc102	1,342.7	8	41	1.52	0.28	1,338.9	7,143	362.43
nrc103	1,213.4	6	31	1.27	0.02	1,213.2	6,811	408.39
nrc104	1,047.5	5	14	1.13	0	1,047.5	3,346	251.16
nrc106	1,228	6	266	3.73	0	1,228	3,443	189.15
nrc107	1,173	6	363	8.09	2.5	1,144.4	2,434	170.71
		Avg =	227	7.74	0.38	Avg =	3,879	245.09

Table 17 - NEWc results for remaining 50-customer problems

On average, 227 iterations were required to find the best solution in a time of 7.74 seconds for the NEWc algorithm results in Table 17. This represents a savings of 98.9% in time to find the best solution when compared to the RTS heuristic VRPTW solutions of Carlton. The overall difference in solution quality from the best solutions was 0.38%. Six of 14 best solutions were matched. A solution using fewer vehicles was found for mixed problem nrc101.

Problem Instance	$Z_t(T)$	Vehs Used(#)	Iter to Best	Time to Best	% Dev	$Z_t(T)$ by Kohl	Iter to Best	Comp Time
nc101	4,862.4	5	0	0.05	0	4,862.4	24	2.07
nc102	4,861.4	5	11	1.02	0	4,861.4	38	14.89
nc103	4,861.4	5	7	1.74	0	4,861.4	34	26.45
nc104	4,855.6	5	226	34.48	0	4,855.6	53	153.56
nc105	4,862.4	5	0	0.06	0	4,862.4	17	1.5
nc106	4,862.4	5	0	0.05	0	4,862.4	19	1.34
nc107	4,862.4	5	0	0.11	0	4,862.4	26	5.43
nc108	4,862.4	5	0	0.22	0	4,862.4	23	3.53
nc109	4,862.4	5	8	2.67	0	4,862.4	18	3.29
nr101	1,545.7	12	5	0.75	0.11	1,544	19	5.86
nr102	1,417.9	10(11)	190	10.89	0.63	1,409	15	3.5
nr103	1,294.8	9	94	2.79	1.72	1,272.9	157	136.09
nr104	1,147.4	7(6)	327	20.68	1.95	1,125.4	676	3,105.7
nr105	1,414.7	9	49	0.98	1.1	1,399.3	56	32.86
nr106	1,293	8	107	2.37	0	1,293	30	14.9
nr107	1,211.1	7	414	16.25	0	1,211.1	255	324.5
nr109	1,290.9	8	376	8.75	0.32	1,286.8	568	523.91
nr110	1,197	7	599	20.59	0	1,197	38	27.13
nr111	1,212.1	7	193	6.74	0.41	1,207.2	4,574	5,054.3
nrc101	1,466.6	9(8)	692	11.36	1.57	1,444	47	26.44
nrc102	1,342.7	8(7)	41	1.52	1.53	1,322.5	1,315	2,209.5
nrc103	1,213.4	6	31	1.27	0.21	1,210.9	53	33.74

nrc104	1,047.5	5	14	1.13	0.16	1,045.8	173	137.61
nrc105	1,361.7	8	156	3.41	0.47	1,355.3	90	70.23
nrc106	1,228	6	266	3.73	0.39	1,223.2	121	135.16
nrc107	1,173	6	363	8.09	2.65	1,142.7	375	623.09
nrc108	1,098.1	6	175	7.24	0	1,098.1	82	598.9
		Avg =	160	6.26	0.49	Avg =	329	491.69

Table 18 - NEWc results, 50-customers

On average, 160 iterations were required to find the best solution in a time of 6.26 seconds for the NEWc algorithm. This represents a savings of 51.4% for number of iterations and 98.7% in time to find the best solution when compared to the exact VRPTW solutions of Kohl. The overall difference in solution quality from the optima was 0.49%. 12 of 27 optimal solutions were matched. Table 19 displays three solutions that violated the time windows constraints in total amounts less than 20 time units and possessed lower objective values.

Problem	$Z_t(T)$	$Penalty_{tw}$	Vehs Used	% Dev
nr102	1,400.6	7	10	-0.6
nr105	1,399.9	3.3	9	0.04
nrc108	1,028.4	9.4	5	-6.35

Table 19 - NEWc infeasible tour results, 50-customers

4.6.3 100-customers

The tables indicate the twelve problems investigated. Desrochers, Desrosiers and Solomon (1992) claim to have solved seven problems to optimality. Carlton (1995, 214-215) used their results for comparison with his

RTS heuristic solutions. Kohl (1995, 179-188) reported to solve all twelve of the investigated problems to optimality and 14 of the 29 Solomon benchmark problems to optimality. However, the optimal schedules for these solutions were not provided except that Kohl's dissertation (1995, 185) indicated that c105 has the same optimal solution as five of the other clustered problems; c101, c102, c106, c107 and c108. Carlton's code was used to identify the optimal tour for the six clustered problems and the best tour for the three radially dispersed problems and the three remaining clustered problems along with their corresponding schedule. These schedules were used to create the PDPTW input data structure. Results are only presented for the NEW algorithms. The INIT algorithm is not competitive timewise for the larger problems. Tables 20-26 use the same format as described previously for the 50-customer problems. 500 iterations were run for the algorithm.

Problem Instance	$Z_t(T)$	Vehs Used	Iter to Best	Time to Best	% Dev	$Z_t(T)$ by Kohl	Iter to Best	Comp Time
nc101	9,827.3	10	0	0.25	0	9,827.3	26	6.4
nc102	9,827.3	10	0	1.91	0	9,827.3	66	98.43
nc104	9,814.8	10	352	166.04	-0.08	9,822.9	57	1,150.8
nc105	9,827.3	10	0	0.51	0	9,827.3	24	6.68
nc106	9,827.3	10	0	0.54	0	9,827.3	33	12.37
nc107	9,826.1	10	75	9.36	-0.01	9,827.3	35	12.4
nc108	9,826.1	10	83	31.49	-0.01	9,827.3	39	26.77
nc109	9,826.1	10	194	110.56	-0.01	9,827.3	35	32.95
		Avg =	88	40.08	-0.01	Avg =	39	168.35

Table 20 - 100-customer NEW results where optimal schedules are known

On average, 88 iterations were required to find the best solution in a time of 40.08 seconds for the NEW algorithm. This represents a savings of 76.2% in time to find the best solution when compared to the exact VRPTW solutions of Kohl. The overall difference in solution quality from the optima was -0.01%. Four of eight optimal solutions were matched. Four "better than" optimal solutions were discovered for problems nc104, nc107, nc108 and nc109 which were reported as optimal by Kohl. These better than optimal solutions result because vehicle demands were modified to account for unloading supplies. The four better than optimal solutions would be infeasible solutions for the VRPTW because of vehicle capacity violations.

Problem Instance	$Z_t(T)$	Vehs Used(#)	Iter to Best	Time to Best	% Dev	$Z_{best}(T)$ Carlton	Iter to Best	Comp Time
nc103	9,829.9	10	25	16.5	0	9,829.9	1,150	324.68
nr101	2,673	20	138	41.1	0.71	2,654.2	1,345	206.6
nr102	2,529.4	18	459	42.7	1.79	2,485	1,863	333.5
nr105	2,494.1	17(16)	217	44.63	5.56	2,362.7	4,770	952.78
		Avg =	209	36.23	2.01	Avg =	2,282	454.39

Table 21 - NEW results for remaining 100-customer problems

On average, 209 iterations were required to find the best solution in a time of 36.23 seconds for the NEW algorithm. This represents a savings of 92% in time to find the best solution when compared to the RTS heuristic VRPTW solutions of Carlton. The overall difference in solution quality from the best solutions was 2.01%. One of four best solutions was matched.

Problem Instance	$Z_t(T)$	Vehs Used(#)	Iter to Best	Time to Best	% Dev	$Z_t(T)$ by Kohl	Iter to Best	Comp Time
nc101	9,827.3	10	0	0.25	0	9,827.3	26	6.4
nc102	9,827.3	10	0	1.91	0	9,827.3	66	98.43
nc103	9,829.9	10	25	16.5	0.04	9,826.3	70	339.72
nc104	9,814.8	10	352	166.04	-0.08	9,822.9	57	1,150.8
nc105	9,827.3	10	0	0.51	0	9,827.3	24	6.68
nc106	9,827.3	10	0	0.54	0	9,827.3	33	12.37
nc107	9,826.1	10	75	9.36	-0.01	9,827.3	35	12.4
nc108	9,826.1	10	83	31.49	-0.01	9,827.3	39	26.77
nc109	9,826.1	10	194	110.56	-0.01	9,827.3	35	32.95
nr101	2,673	20(18)	138	41.1	1.34	2,637.7	90	142.38
nr102	2,529.4	18(17)	459	42.7	2.55	2,466.6	32	97.56
nr105	2,494.1	17(15)	217	44.63	5.89	2,355.3	490	2,331.1
		Avg =	128	38.8	0.8	Avg =	83	354.8

Table 22 NEW results, 100-customers

Overall, 128 iterations were required, on average, to find the best solution in a time of 38.8 seconds for the NEW algorithm. This represents a savings of 89.1% in time to find the best solution when compared to the exact VRPTW solutions of Kohl. The overall difference in solution quality from the optima was 0.8%. 4 optimal tours were matched. The NEW algorithm found better results for the clustered problems c104, c107, c108 and c109 because of the modified PDPTW demands. One better infeasible solution was found using this algorithm for problem nr105. The travel time was 2,405.5 with a 10.5 time windows violation. The tour used 15 vehicles and was 2.13% from the optimal solution.

Problem Instance	$Z_t(T)$	Vehs Used	Iter to Best	Time to Best	% Dev	$Z_t(T)$ by Kohl	Iter to Best	Comp Time
nc101	9,827.3	10	0	0.24	0	9,827.3	26	6.4
nc102	9,827.3	10	0	2	0	9,827.3	66	98.43
nc104	9,814.8	10	441	459.07	-0.08	9,822.9	57	1,150.8
nc105	9,827.3	10	0	0.5	0	9,827.3	24	6.68
nc106	9,827.3	10	0	0.55	0	9,827.3	33	12.37
nc107	9,826.1	10	31	4.03	-0.01	9,827.3	35	12.4
nc108	9,826.1	10	30	12.73	-0.01	9,827.3	39	26.77
nc109	9,826.1	10	82	34.22	-0.01	9,827.3	35	32.95
		Avg =	73	64.17	-0.01	Avg =	39	168.35

Table 23 - 100-customer NEWc results where optimal schedules are known

On average, 73 iterations were required to find the best solution in a time of 64.17 seconds for the NEWc algorithm. This represents a savings of 61.9% in time to find the best solution when compared to the exact VRPTW solutions of Kohl. The overall difference in solution quality from the optima was -0.01%. Four of eight optimal solutions were matched. Again, four "better than" optimal solutions were discovered for problems nc104, nc107, nc108 and nc109 which were reported as optimal by Kohl. These better than optimal solutions result because vehicle demands were modified to account for unloading supplies. The four better than optimal solutions would be infeasible solutions for the VRPTW because of vehicle capacity violations.

On average, 154 iterations were required to find the best solution in a time of 33.05 seconds for the NEWc algorithm. This represents a savings of

92.7% in time to find the best solution when compared to the RTS heuristic VRPTW solutions of Carlton. The overall difference in solution quality from the best solutions was 1.02%. One of four best solutions was matched and a better solution was found for nr101.

Problem Instance	$Z_t(T)$	Vehs Used(#)	Iter to Best	Time to Best	% Dev	$Z_{best}(T)$ Carlton	Iter to Best	Comp Time
nc103	9,829.9	10	25	17.05	0	9,829.9	1,150	324.68
nr101	2,645.6	20	212	33.75	-0.32	2,654.2	1,345	206.6
nr102	2,593.9	19(18)	73	23.27	4.38	2,485	1,863	333.5
nr105	2,363	16	308	58.13	0.01	2,362.7	4,770	952.78
		Avg =	154	33.05	1.02	Avg =	2,282	454.39

Table 24 - NEWc results for remaining 100-customer problems

Problem Instance	$Z_t(T)$	Vehs Used(#)	Iter to Best	Time to Best	% Dev	$Z_t(T)$ by Kohl	Iter to Best	Comp Time
nc101	9,827.3	10	0	0.24	0	9,827.3	26	6.4
nc102	9,827.3	10	0	2	0	9,827.3	66	98.43
nc103	9,829.9	10	25	17.05	0.04	9,826.3	70	339.72
nc104	9,814.8	10	441	459.07	-0.08	9,822.9	57	1,150.8
nc105	9,827.3	10	0	0.5	0	9,827.3	24	6.68
nc106	9,827.3	10	0	0.55	0	9,827.3	33	12.37
nc107	9,826.1	10	31	4.03	-0.01	9,827.3	35	12.4
nc108	9,826.1	10	30	12.73	-0.01	9,827.3	39	26.77
nc109	9,826.1	10	82	34.22	-0.01	9,827.3	35	32.95
nr101	2,645.6	20(18)	212	33.75	0.3	2,637.7	90	142.38
nr102	2,593.9	19(16)	73	23.27	5.16	2,466.6	32	97.56
nr105	2,363	16(15)	308	58.13	0.33	2,355.3	490	2,331.1
		Avg =	100	53.8	0.47	Avg =	83	354.8

Table 25 - NEWc results, 100-customers

On average, 100 iterations were required to find the best solution in a time of

53.8 seconds for the NEWc algorithm. This represents a savings of 84.8% in time to find the best solution when compared to the exact VRPTW solutions of Kohl. The overall difference in solution quality from the optima was 0.47%. Four optimal tours were matched. The NEW algorithm found better results for the clustered problems c104, c107, c108 and c109 because of the modified PDPTW demands as discussed in Section 3.3.

4.7 CONCLUSIONS

Limiting the search to only precedence viable solutions allows for use of an expensive SPI neighborhood search scheme. The dominance of the precedence and coupling constraints is critical to developing appropriate search strategies. The algorithms are compared to the current best known optimal and heuristic approaches to the VRPTW. The NEW algorithms consistently return solutions within one percent, on average, in a fraction of the computational effort required by the other algorithms. Feasible solutions are found for all problem instances along with several infeasible tours. The infeasible solutions obtained are more than workable in a real world scenario.

Optimal or near-optimal solutions are obtained for the modified problem instances in real-time despite the addition of increased information to transform the VRPTW data sets into PDPTW data sets. Plus, not all the optimal schedules were known for all the problem instances. Setting up the input data structure for

the best solution as compared to the optimal tour displayed problems with the search. Often times, the best solution competed with the hidden optimal tour to hinder the search trajectory. This led the search to find a sub-optimal tour. However, NEW and NEWc displayed its strength by being able to explore infeasible regions of the solution space and find competitive solutions and even infeasible tours with reduced travel times.

The PDPTW solutions obtained in this research were compared to the optimal solutions or the best solutions that Carlton's code could generate for the VRPTW. The solutions that are reported by the PDPTW cannot be reported as optimal for a couple reasons. First, the PDPTW allows for both loading and unloading of supplies. Routes that finished "early" in the VRPTW problem instances because of capacity limitations do not necessarily encounter that problem in the PDPTW. Vehicles may be able to continue gathering and delivering supplies until the planning horizon requires the vehicle to return to the depot for the PDPTW. However, these routes generated by the PDPTW would violate the vehicle capacity constraints as evidenced in the four 100-customer clustered problems from section 4.6.3. Finally, the PDPTW algorithm presented in this research is a metaheuristic and not an exact algorithm. This research can only report its solutions as the current best solution found until other heuristic procedures find better solutions or an exact method is developed to validate the

results of this work.

Modifying the candidate list for the NEWc algorithm expedites the overall search process. By altering the search trajectory NEWc was able to uncover more infeasible tours with reduced travel times.

The expectation for the INIT algorithm was to achieve comparable results for small problems. However, as the problem instances grew, solutions were of poorer quality than the NEW and NEWc algorithms. The results also reconfirm the results of other researchers, that it is better to start with a feasible initial tour.

Chapter 5

The Generalized Precedence Scenarios

In real world situations, a supplier can provide support for several customers and retail stores can receive supplies from several different companies. In these scenarios the order for picking up supplies or making deliveries may not be important. The objective will be to minimize the total travel time for distribution to assist in keeping the price of the product or service low. Serial precedence will be established if there exists some predefined ordering.

This chapter will discuss how to transform three *generalized*-PCRPTWs into PDPTWs. Modifications to the input data structure will be presented and representative problems used to illustrate the transformation of a single supplier supporting several delivery locations, several suppliers supporting a single delivery location and serial precedence where a predefined ordering is known for a mixture of suppliers and delivery locations. For these scenarios, recall the following notation:

$T \equiv$ the current value of the travel time or the objective function value,
$t_{i,j} \equiv$ the travel time from node i to node j, with
$0 \equiv$ to be the depot node.

The final example, a true PCRPTW, will demonstrate how to modify the input data structure when the three mentioned scenarios are combined.

5.1 The Single Supplier Supporting Several Delivery Locations.

This scenario can be transformed to the simple pairwise precedence scenario or PDPTW by adding dummy supply nodes to match the total number of delivery nodes. The following digraphs, figures 4 and 5, and supporting input data illustrate this procedure.

The basic model is

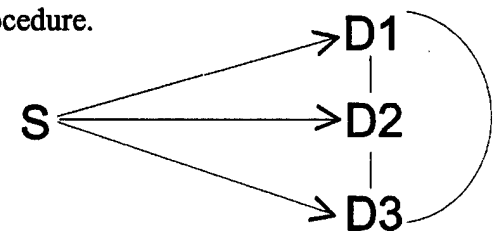

Figure 4 - The generalized precedence digraph

node	x_i	y_i	demand	early	late	service	pred	succ
S	x_S	y_S	d_S	e_S	l_S	s	0	D1, 2 or 3
D1	x_{D1}	y_{D1}	$-d_{D1}$	e_{D1}	l_{D1}	s_{D1}	S	0
D2	x_{D2}	y_{D2}	$-d_{D2}$	e_{D2}	l_{D2}	s_{D2}	S	0
D3	x_{D3}	y_{D3}	$-d_{D3}$	e_{D3}	l_{D3}	s_{D3}	S	0

Table 26 - The supporting input data structure for the single-to-many model

In this model, $d_s = \left|\Sigma_{i=1}^{3} d_{Di}\right|$. The supply node, S, does not have a clearly defined immediate successor. The immediate successor must be one of the three delivery locations. However, the supply node must be visited prior to visiting any of the three delivery locations.

Applying the transformation, the modified model becomes the digraph in Figure 5. The directed arcs indicate precedence while the edges display the

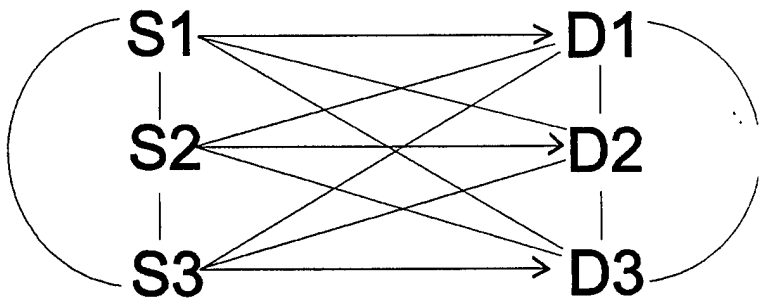

Figure 5 - The transformed digraph

random ordering among the delivery locations that the algorithm must sort out. The associated transformed input data is reflected in Table 27.

node	x_i	y_i	demand	early	late	service	pred	succ
S1	x_S	y_S	d_S	e_S	l_S	s	0	D1
D1	x_{D1}	y_{D1}	$-d_{D1}$	e_{D1}	l_{D1}	s_{D1}	S1	0
S2	x_S	y_S	0	e_S	l_S	0	0	D2
D2	x_{D2}	y_{D2}	$-d_{D2}$	e_{D2}	l_{D2}	s_{D2}	S2	0
S3	x_S	y_S	0	e_S	l_S	0	0	D3
D3	x_{D3}	y_{D3}	$-d_{D3}$	e_{D3}	l_{D3}	s_{D3}	S3	0

Table 27 - The supporting input data structure for the transformed model

The dummy nodes will identically match the original supply node except that the service time, *s*, will be set to zero.

The objective for the algorithm is to minimize total travel time. Thus, the algorithm will choose one of the six contiguous orderings of the supply nodes ... S1 - S2 - S3 ... before it selects one of the corresponding delivery locations. No other supply nodes or delivery nodes will be interspersed amongst this ordering

because that would increase the total travel time. Specifically, a sample initial tour ordering could be represented as

... □ S1 D1 □ S2 D2 □ S3 D3 □ S4 D4 □ ... □ S*n* D*n* □ ...

The SPI neighborhood search will determine if it is more profitable to combine the node pair (S1, D2) with node pair (S2, D2) or, say, node pair (S4, D4). The total travel time will be modified, for example, choosing the former tour ... □ S1 S2 D2 D1 □ ... decreases the initial total travel time by

$$T - t_{0,S2} - t_{D2,0} + t_{D2,D1}$$

SPI will determine the appropriate ordering for the supply nodes and delivery nodes to cause the greatest reduction in travel time. The supply nodes will be coupled together. Similarly, to insert the node pair (S4, D4) onto route 1, ... □ S1 S4 D4 D1 □ ..., adjusts the initial travel time by

$$T - t_{0,S4} - t_{D4,0} + t_{S1,S4} + t_{D4,D1}$$

Adding the time to travel "between the supply nodes" makes this perturbation not as likely. In a clustered problem with S4 located close to S1 and D4 close to D1, this second case may prove more beneficial only if node D2 is located further away from the depot than D4, i.e.,

$$t_{0,S2} \approx t_{0,S4}, t_{S1,S4} \approx 0, t_{D4,D1} \approx 0 \text{ so that } t_{D4,0} \leq t_{D2,0} - t_{D2,D1}.$$

This transformation reveals if additional vehicles will be required for that supplier to support those corresponding deliveries within the planning horizon.

Obviously, the node pairs (S1, D1), (S2, D2) and (S3, D3) will end up on different routes only if they cannot all be supported by the same vehicle.

Example #1. The data set SINGLEC modifies problem set nc101, the first 25-customer clustered problem presented by Solomon. The first route in the optimal tour is ordered

5 - 3 - 7 - 8 - 10 - 11 - 9 - 6 - 4 - 2 - 1 and 26

where node 26 was modeled after node 1 to provide for pairwise precedence. The first route precedence is altered to provide a single supplier, supporting 3 delivery locations, or

7 - 8, 10, 11

where node 7 supplies delivery locations 8, 10 and 11. Dummy nodes 27 and 28 modeling node 7 were added. The best feasible tour, like the original problem instance, was the initial tour. The resulting schedule is depicted in table 28. The shading is used to delineate the end of the route and the return to the depot. Routes will be designated by the vehicle identification number located at the end of the route. In this instance, the first route is designated route 29; the second, route 30; and the last, route 31. The optimal travel time of this tour is 2441.3. The cost of the tour is 2882 which means drivers wait for 440.7 minutes. The length of the tour is 1049.2. The search found only a total of 19 feasible tours and 100 different tours because the average time window length of 51.7 for a

planning horizon of 1236 indicates tight time windows and few

node i	arrival[i]	departure[i]	late[i]
0	0	0	1,236
5	15.1	15.1	55
3	106.1	106.1	146
*27	198.1	198.1	225
*28	198.1	198.1	225
*7	198.1	198.1	225
8	290.9	290.9	324
10	384.5	384.5	410
11	477.5	477.5	505
9	570.6	570.6	605
6	662.8	662.8	702
4	755	755	776.4
2	848.6	848.6	870
1	940.6	940.6	967
26	940.6	940.6	967
29	1,049.2	0	1,236
13	30.8	30.8	54
17	124.8	124.8	148
18	217.8	217.8	250
19	312.8	312.8	345
15	407.8	407.8	429
16	502.8	502.8	528
14	594.8	594.8	620
12	687.8	687.8	721
30	815.8	0	1,236
20	10	10	49
24	105	105	144

25	197	197	224
23	291.2	732	777
22	825	825	873
21	917	917	965
31	1,017.1	0	1,236

* These 3 nodes can occur in any order.

Table 28 - Optimal Tour Data for SINGLEC

feasible tours. The 19 feasible tours found is accurate for only the transformed instance, not the original problem.

SINGLEC using INIT starts with an initial marginally infeasible tour with a tour cost of 5250.3 and a travel time of 2522.0 for 5 vehicles. INIT uses Solomon's I1 insertion algorithm to generate the marginally infeasible tour . The best feasible and optimal tour was found on the 5th iteration. The only difference in the tour is it provides a different ordering - 28, 27, 7 - for the three similar nodes. This search found a total of 9 feasible tours out of 100 different tours. It encountered a lesser number of feasible tours because the search starts with an infeasible tour and expends effort correcting the time windows violations.

Example #2. SINGLER is a radially dispersed problem where a single supplier supplies 3 delivery locations. This problem is modified from nr102. This problem was selected because it provides infeasible tours with reduced travel times and the optimal tour was identified by the NEWc algorithm, but not the

NEW algorithm. The pairwise precedence for the last route in table 29 was modified

node i	arrival[i]	departure[i]	late[i]
...			
36	215.5	0	230
*29	22.3	22.3	189.5
*3	22.3	22.3	82
*30	32.3	32.3	113.7
9	47.3	97	107
20	118.1	126	136
1	152.4	152.4	204
37	177.6	0	230

* Nodes 29, 3 and 30 can occur in any order.

Table 29 - Optimal Tour Data for SINGLER

from 3 - 9 and 20 - 1 to a generalized precedence of 3 supplying 9, 20 and 1. Dummy nodes 29 and 30 modeling node 3 were added to the problem. Dummy nodes 26 - 28 were needed for pairwise precedence for the three routes from the optimal tour with an odd number of customers.

NEWc finds the optimal solution for SINGLER. The best feasible tour was found on the 17th iteration, much faster than the 281 iterations required for the nr102 version. The resulting schedule for the final route is depicted in table 29. The cost of the tour is 1256 with an optimal travel time of 797.7 using 7 vehicles. A super optimal tour was found on the 87th iteration. The resulting

schedule is:

node i	arrival[i]	departure[i]	late[i]
0	0	0	230
14	32	32	42
16	53.1	75	85
8	108	108	105.0*
17	131.9	157	167
5	177	177	199
28	177	177	199
33	207.6	0	230
18	15.8	87	97
26	87	87	97
6	108.1	108.1	109
13	125.1	159	169
38	180.1	0	230
7	21.2	21.2	52
11	46.2	67	77
19	84	84	86
10	109	124	134
39	159.4	0	230
21	18	18	50
23	46	68	78
24	109.6	153	163
12	178	178	205
42	203	0	230
2	18	18	48
15	41	61	71
22	86.8	97	107
4	121.1	149	159

25	169	172	182
27	172	172	182
43	215.5	0	230
29	22.3	22.3	189.5
30	22.3	22.3	113.7
3	22.3	22.3	82
9	47.3	97	107
20	118.1	126	136
1	152.4	152.4	204
45	177.6	0	230

Table 30 - Infeasible Tour Data for SINGLER

The travel time of this tour is 771.2 using 6 vehicles. This provides a savings of over 3% in total travel and one vehicle. The time windows violation is only 3 units for one customer which is negligible. Table 30 illustrates one of several the infeasible tours discovered. The search found a total of 84 feasible tours and 100 different tours. The average time window length is 38.7 units and the planning horizon is 230 units.

NEW found the same feasible solution later in the search for SINGLER than the original nr102. This search trajectory does not enable the routine to find the optimal solution. The same infeasible tours were identified during the search.

INIT found the optimal solution on the 44th iteration. INIT also did not find the optimal tour for the original nr102 instance. The added structure

afforded by the transformation helped the search discover the optimal. The search still only found a total of 57 feasible tours out of 100 different tours. Infeasible tours were not identified.

Example #3. SINGLEM is a radially dispersed and clustered mixed problem that has one supplier supplying 3 delivery locations. It is modified from nrc103. The original ordering for the modified route is

7, 6, 8, 5, 3, 1, 4 and 2.

The precedence was modified to

7 - 1; 4 - 2; and 6 - 8, 5 and 3.

interspersing the generalized precedence ordered subset (*g*POS) between the *pairwise*-POS. Dummy nodes 27 and 28 were added to model node 6.

SINGLEM using NEW found the optimal tour in the same number of iterations as the original problem instance. Table 31 shows the resulting schedule for the altered route.

node i	arrival[i]	departure[i]	late[i]
...	...	...	...
7	35.3	35.3	166.9
*27	48.3	95	125
*6	95	95	105.2
*28	105	105	125
8	110.8	110.8	121
5	127.8	127.8	189

3	139.8	139.8	190
1	152.8	152.8	191
4	169.8	169.8	171
2	185.1	185.1	199
31	225.9	0	240

* Again, these three can occur in any order.

Table 31 - Optimal Tour Data for SINGLEM

This problem shows the ability of the code to determine the correct ordering when the pairs are interspersed amongst each other.

SINGLEM using NEWc also found the optimal tour on the 4th iteration. The only difference was the ordering of the 3 similar nodes. In this instance, they were ordered 6, 27 and 28.

SINGLEM using INIT also found the optimal tour early in the search - by the 8th iteration.

5.2 Several Suppliers Supporting a Single Delivery Location.

This scenario can be approached in the same manner as the single-to-many problem instance by adding a delivery node for each additional supply node that must be visited before making the delivery. The dummy nodes modeling the delivery node will have no demand or service time requirements and the appropriate pairwise precedence relationship will be established with a corresponding supply node. The algorithm will determine the ordering of

suppliers that will minimize the travel time of making the group delivery. Three example problems will be used to illustrate this process.

Example #1. MANYC has 3 suppliers supporting a single delivery location. The problem was modified from nc104, a problem instance with large time windows; the average time window length (*atwl*) of 699.2 comprises over 56% of the planning horizon length of 1236. Because of the large time windows, all of the tours found are feasible. Plus, the neighborhood search methodologies will take longer because of the increased number of moves that they must examine in order to determine the best move to make.

To further test the code, the ordering for the route

7, 8, 11, 9, 6, 4, 2, 1, 3 and 5.

is assigned the precedence

7 - 9; 6 - 4; 3 - 5; and 8, 11, 2 - 1.

where nodes from the *g*POS are broken up and interspersed among the *pairwise*-POSs.

NEW and NEWc required 5 more iterations to solve MANYC than the 21 iterations it took in the original problem instance. The resulting schedule for this modified route is:

node i	arrival[i]	departure[i]	late[i]
0	0	0	1,236
7	16	16	510.6
8	108.8	155	324
11	348.1	48	505
9	541.1	41.1	605
6	633.3	33.3	689.8
4	725.5	27	782
2	820.6	20.6	1,035
*1	912.6	1,002.6	1,127
*27	1,002.6	1,002.6	1,127
*28	1,002.6	1,002.6	1,127
3	1,006.2	1,006.2	1,039
5	1,097.2	1,097.2	1,130
29	1,202.3	0	1,236
...	...	...	...

* can be ordered 1 of 6 ways.

Table 32 - Optimal Tour Data for Route 29 of MANYC

Solving MANYC using INIT produced optimal results. However, it took over three times longer to find the optimal tour; 76 versus 31 iterations, as it took in the original PDPTW. In this instance, the additional nodes forced the procedure to expend more effort to clean up the initial tour and altered the search trajectory enough to require more iterations to determine the optimal tour. The optimal tour was still discovered fairly early in the search.

Example #2. MANYR is a radially dispersed 25-customer problem modified

from nr105. This had been a tough problem instance to find the optimal solution to until the NEW and NEWc routines were developed using the sequential *pairwise*-POS time-space insertion algorithm to create the initial tour. The older algorithm, called BUILD, placed a single *pairwise*-POS on a route for the initial tour requiring dedicated computer effort initially to reduce the number of vehicles This created problems, as discussed in Chapter 4, whereby the search blew by the optimal. INIT cannot even find a feasible solution. This instance has few feasible solutions because the time windows are tight, about 11% of the planning horizon (25.7 out of 230).

This problem uses 2 many-to-single operations. The first has nodes 19, 11, 10 and 20 supporting customer 1. A separate route has nodes 2 and 15 supporting customer 13. Four additional dummy nodes, nodes 27 - 30, are required for the transformation.

The added structure provided by making these modifications resulted in a drastic reduction in the number of iterations to find the optimal tour. NEW finds the optimal tour on the 9th iteration as compared to the 23 iterations in the original problem instance. NEWc requires 61 iterations, way down from the 390 iterations for nr105. The resulting schedule is

node i	arrival[i]	departure[i]	late[i]
0	0	0	230
7	21.2	71	87
18	91	91	107
8	111.4	111.4	115
17	135.3	147	177
31	187.4	0	230
2	18	40	70
15	63	63	81
*30	93	149	179
*13	149	149	179
32	170.1	0	230
5	20.6	24	32
14	54	54	62
16	75.1	75.1	91
6	103.1	103.1	119
33	124.2	0	230
19	32	66	96
11	83	83	87
10	104.1	114	144
20	139.8	139.8	146
*1	166.2	166.2	181
*29	176.2	176.2	181
*28	176.2	176.2	181
*27	176.2	176.2	181
34	191.4	0	230
21	18	52	60
23	80	80	88
22	101.1	101.1	117
4	125.2	139	169

35	174	0	230
12	15	53	81.6
9	88.4	88.4	117
3	113.4	113.4	136
24	137.5	143	173
25	168	168	186
26	168	168	186
36	211.5	0	230

Table 33 - Optimal Tour Data for MANYR

The cost of the tour is lowered to 1058. The length of the tour is 211.5. The optimal travel time of this tour is 780.5. Only 20 feasible tours were found out of 100 different tours examined.

MANYR using INIT did not find the optimal tour. The search progressed much better than INIT did in solving nr105. MANYR search found 15 feasible tours out of the 100 different tours investigated. This is due to the added structure provided by the transformation. The best feasible tour was found on the 31st iteration. This feasible solution is better than the infeasible solution found for nr105. The best travel time of this tour is 801.6.

Example #3. MANYM is a mixed problem modified from nrc108 that has 4 suppliers supporting a single delivery location. To test the versatility of the code, the route

12, 14, 17, 16, 15, 13, 9, 11, 10

is assigned the ordering

12 - 17; 15 - 9; and 14, 16, 13, 11 - 10.

This places a node from the *g*POS between each node from the two *pairwise*-POSs. Dummy nodes 26 - 28 are added to model node 10.

All three procedures were able to find the optimal tour earlier in its search for MANYM. NEW and NEWc required only 5 iterations instead of 16 iterations to find the optimal tour. INIT found the optimal tour in 27 instead of 39 iterations. The resulting schedule for the second of the three routes in the optimal tour is displayed in table 34. The optimal cost of the tour is 544.5. The average time window length was over 38% of the planning horizon length. Thus, there were a considerable number of feasible tours to be discovered. The NEW procedures found better than 50% feasible tours while INIT found slightly less than half of the tours feasible.

node i	arrival[i]	departure[i]	late[i]
...	...	...	...
12	32.3	32.3	148
14	45.3	45.3	144
17	60.3	60.3	189
16	75.3	75.3	141
15	87.3	87.3	134.6
18	103.1	103.1	144
9	118.1	118.1	154
11	133.4	133.4	149

*10	145.4	145.4	161
*26	155.4	155.4	161
*28	155.4	155.4	161
*27	155.4	155.4	161
31	187	0	240
...	...	...	...

* can be ordered many different ways.

Table 34 - Optimal Tour for MANYM

5.3 The Serial Precedence Model

The serial precedence scenario can also be modeled as a PDPTW by mirroring all intermediary nodes. For example, require supplier 1 (S1) to visit supplier 2 (S2) prior to visiting supplier 3 (S3) before their joint delivery to D1. This defines the precedence ordered subset (*s*POS) S1 - S2 - S3 - D1. The transformation provides for a modified POS of S1 - S2a - S2b - S3a - S3b - D1. The original input data structure is outlined in table 35.

node	x_i	y_i	demand	early	late	service	pred	succ
S1	x_{S1}	y_{S1}	d_{S1}	e_{S1}	l_{S1}	s_{S1}	0	S2
S2	x_{S2}	y_{S2}	d_{S2}	e_{S2}	l_{S2}	s_{S2}	S1	S3
S3	x_{S3}	y_{S3}	d_{S3}	e_{S3}	l_{S3}	s_{S3}	S2	D1
D1	x_{D1}	y_{D1}	d_{D1}	e_{D1}	l_{D1}	s_{D1}	S3	0

Table 35 - Initial input data for the serial precedence model

The structure is modified to establish the pairwise relationship in table 36.

node	x_i	y_i	demand	early	late	service	pred	succ
S1	x_{S1}	y_{S1}	d_{S1}	e_{S1}	l_{S1}	s_{S1}	0	S2a
S2a	x_{S2}	y_{S2}	d_{S2}	e_{S2}	l_{S2}	0	S1	0
S2b	x_{S2}	y_{S2}	0	e_{S2}	l_{S2}	s_{S2}	0	S3a
S3a	x_{S3}	y_{S3}	d_{S3}	e_{S3}	l_{S3}	0	S3b	0
S3b	x_{S3}	y_{S3}	0	e_{S3}	l_{S3}	s_{S3}	0	D1
D1	x_{D1}	y_{D1}	d_{D1}	e_{D1}	l_{D1}	s_{D1}	S3b	0

Table 36 - The transformed input data structure for the serial precedence model

This transformation would have to occur because the current SPI neighborhood search scheme only accounts for pairs/two members in the search. SPI would move a predecessor and its corresponding successor, but not the other members of the POS creating coupling constraint violations. S2a is the original supply node S2. S2b is the dummy node added to the data set.

It is possible for coupling constraints to be violated in the transformed problem after the formation of the initial tour. However, the search process would sort out this "hidden" violation and ensure that the serial precedence is established. For example, lets say part of the initial tour is depicted below:

... □ S1 S2a □ S2b S3a □ S3b D1 □ ... □ S4 D4 □ S5 D5 □ ...

Combining the routes

... □ S1 S2a □ S2b S3a S3b D1 □ ... □ S4 D4 □ S5 D5 □ ...

is more desirable than to combine the last two depicted routes

... □ S1 S2a □ S2b S3a □ S3b D1 □ ... □ S4 D4 S5 D5 □ ...

because of the same rationale provided in section 5.1.

If at the end you do not find the POS S1 - S2a - S2b - S3a - S3b - D1 on the same route, the problem has no feasible solution. However, the "best tour" generated by the algorithm could possibly reveal the POS on the same route if the time windows violations are tolerable and generate the minimum travel time.

Example #1. The first example modifies clustered problem nc105 by modifying the precedence for route 3 (**) below. The optimal ordering for the route is

5, 3, 7, 8, 10, 11, 9, 6, 4, 2, 1 and 26.

This route is assigned the serial precedence of

5 - 3; 7 - 8; 10 - 11; and 9 - 6 - 4 - 2 - 1.

Nodes 9, 6, 4 and 2 are all supplying delivery node 1. Dummy nodes are prepared for (6, 26), (4, 27) and (2, 28).

NEW and NEWc also had the optimal tour for SERIAL1 as the initial tour. The optimal travel time of this tour is 2441.3. INIT required 20 iterations to find its best feasible tour with a travel time of 2465.2. The best solution that INIT could find for the original problem instance had a travel time of 2483. INIT performed better for this instance, again, because of the increased ordering information. The optimal ordering for the modified route is depicted in table 37.

node i	arrival[i]	departure[i]	late[i]
...	...	...	...
5	15.1	15.1	95
3	106.1	106.1	186
7	198.1	198.1	253
8	290.9	290.9	359
10	384.5	384.5	436
11	477.5	477.5	533
9	570.6	570.6	640
6	662.8	662.8	743
26	662.8	662.8	717.8
4	755	755	810
27	755	755	799.4
2	848.6	848.6	893
28	848.6	848.6	893
1	940.6	940.6	994
31	1,049.2	0	1,236

Table 37 - Optimal Tour Data for SERIAL1

Example #2. SERIAL2 modifies the radially dispersed problem nr108. This problem instance places part of the *serial*-POS between a *pairwise*-POS. The optimal ordering of the fourth route depicted in the table 38 below has the ordering

2, 15, 14, 16, 17, 5, 6 and 13

The *s*POS for this fourth route will place nodes 15 and 14 between node pair (2, 16). This creates the precedence ordering of

2 - 16; 15 - 14 - 17 - 5; and 6 - 13

with nodes 15, 14 and 17 supplying delivery node 5. Dummy nodes (14, 27) and (17, 28) were added to complete the transformation.

Both NEW and NEWc found the optimal solution after 13 iterations where it required 56 and 196 iterations, respectively, for the original problem instance. The additional structure provided by serial relationship aided in the discovery of the optimal tour.

node i	arrival[i]	departure[i]	late[i]
...	...	...	...
**2	18	18	154.9
15	41	51	81
27	76.8	76.8	156.7
14	86.8	86.8	187
16	97.9	97.9	190
28	119	119	179
17	129	129	189
5	139	139	199
6	159	159	162
13	176	176	179
32	197.1	0	230

Table 38 - Optimal Tour Data for Route 32 of SERIAL2

This problem illustrates one of the problems that may arise with this transformation - multiple solutions. The ordering for route 4 may also be written as

2, 15, 14, 27, 16, 28, 17, 5, 6, 13.

The user needs to be familiar with the dummy nodes used to determine the actual route being represented.

INIT required 34 iterations to find the optimal solution for SERIAL2. INIT also found the optimal tour in less iterations for the modeled problem, nr108. This indicates that the marginally infeasible initial tour provided suitable start point for the search to commence.

Example #3. SERIAL3 is a mixed radially dispersed and clustered problem modified from nrc105. This problem interposes the entire *serial*-POS between a *pairwise*-POS. This problem instance was also selected because it generates several infeasible tours with a lower objective value. The optimal solution's fourth route is ordered

12, 14,16, 15, 13 and 17

SERIAL3 imposes the precedence relationships

12 - 17 and 14 - 16 - 15 - 13.

where nodes 14, 16, and 15 all supply delivery node 13. Dummy nodes for (16, 27) and (15, 28) were added. The entire optimal tour is represented in table 39 for comparison with the infeasible tour discovered.

node i	arrival[i]	departure[i]	late[i]
0	0	0	240
11	33.5	69	79
9	84.3	101	111

10	116	123	144
26	123	123	144
30	164.6	0	240
2	30.8	30.8	139.9
5	50.9	50.9	160
3	62.9	64	178
1	77	77	191
8	98.1	101	107.2
6	116.8	116.8	123
7	129.8	129.8	144
4	146.8	151	161
33	197	0	240
19	40	58	68.6
23	74.4	75	85
18	91	91	98.3
22	111.7	111.7	119
20	123.7	123.7	144.9
21	143.8	143.8	165
25	158.8	171	173.6
24	191.4	191.4	194
40	236.4	0	240
**12	32.3	32.3	152
14	45.3	45.3	97
16	62.3	62.3	114
27	62.3	62.3	66
15	74.3	74.3	78
28	74.3	74.3	78
13	90.1	152	162
17	173.1	173.1	180
41	223.4	0	240

Table 39 - Optimal Tour Results for SERIAL3

NEW and NEWc both found the optimal tour on the sixth iteration, compared to the 7 and 60 iterations required for the original problem instance. The cost of the tour is 821 with a corresponding travel time of 661.3. There is a relatively large amount of time that drivers must wait in this problem, 159.7 units. This problem structure provides the opportunity for infeasible tours with reduced total travel times. The customers on the first route in table 40 are inserted on other routes. The search also finds the same super-optimal tour generated by problem nrc105. The best overall travel time tour was found on the 114th iteration. Again, infeasible tours are found during the second half of the search. Vehicle reduction to an infeasible tour is not permitted until half the search is completed. The best overall tour for 3 vehicles is displayed in table 40.

node i	arrival[i]	departure[i]	late[i]
0	0	0	240
2	30.8	30.8	139.9
5	50.9	50.9	160
3	62.9	64	178
1	77	77	191
8	98.1	101	107.2
6	116.8	116.8	123
7	129.8	129.8	144
4	146.8	151	161
33	197	0	240
19	40	58	68.6

23	74.4	75	85
18	91	91	98.3
22	111.7	111.7	119
20	123.7	123.7	144.9
21	143.8	143.8	165
25	158.8	171	173.6
24	191.4	191.4	194
40	236.4	0	240
12	32.3	32.3	152
14	45.3	45.3	97
27	62.3	62.3	66
16	72.3	72.3	114
15	74.3	74.3	78
28	74.3	74.3	78
11	90.3	90.3	79.0*
9	105.6	105.6	111
10	120.6	123	144
26	123	123	144
13	140	152	162
17	173.1	173.1	180
41	223.4	0	240

Table 40 - Infeasible Tour Results for SERIAL3

The tour cost is further reduced to 656.8 and the corresponding travel time is 603.4. The time windows violation is only 11.3 units for one customer. This search also found three workable tours with better travel times than the optimal. the search found a total of 91 feasible tours out of 200 different tours. These feasible tours were all found during the first half of the search. The second half

of the search explored infeasible solutions. There does not exist a feasible solution for three vehicles. Once a vehicle has been eliminated during the search, it is never returned.

SERIAL3 using INIT found the optimal tour on the 96th iteration and the best tour on the 114th iteration.

5.4 Conclusions

Despite the fact that the transformations enlarge the problem, the transformations provide added definition to the problem instances helping the search determine the optimal tour. This increased knowledge about the ordering of the customers even helped INIT find feasible solutions and optimal tours in problems that it routinely had difficulty finding feasible tours and the optimal solution.

The three scenarios provided can be combined and modified to handle the most generalized of precedence scenarios. For example, given three suppliers supporting three delivery locations in the following manner:

a. supplier, S1, supplies customer D1,

b. suppliers S1 and S3 supply customer D2 and S3 must be visited after S1, and

c. suppliers S2 and S3 supply customer D3.

The corresponding input data structure is shown in table 19 where

$\Sigma_i d_{Si} + \Sigma_i d_{Di} = 0.$

node	x_i	y_i	demand	early	late	service	pred	succ
S1	x_{S1}	y_{S1}	d_{S1}	e_{S1}	l_{S1}	s_{S1}	0	D1 or S
D1	x_{D1}	y_{D1}	$-d_{D1}$	e_{D1}	l_{D1}	s_{D1}	S1	0
S2	x_{S2}	y_{S2}	d_{S2}	e_{S2}	l_{S2}	s_{S2}	0	D3
D2	x_{D2}	y_{D2}	$-d_{D2}$	e_{D2}	l_{D2}	s_{D2}	S3	0
S3	x_{S3}	y_{S3}	d_{S3}	e_{S3}	l_{S3}	s_{S3}	S1	D2 or D
D3	x_{D3}	y_{D3}	$-d_{D3}$	e_{D3}	l_{D3}	s_{D3}	S2 or S3	0

Table 41 - Input Data for Generalized Scenario

With the addition of four dummy nodes, this generalized scenario can be transformed into a PDPTW. The transformed data set would be

node	x_i	y_i	demand	early	late	service	pred	succ
S1	x_{S1}	y_{S1}	d_{S1}	e_{S1}	l_{S1}	s_{S1}	0	D1
S1a	x_{S1}	y_{S1}	0	e_{S1}	l_{S1}	0	0	S3
D1	x_{D1}	y_{D1}	$-d_{D1}$	e_{D1}	l_{D1}	s_{D1}	S1	0
S2	x_{S2}	y_{S2}	d_{S2}	e_{S2}	l_{S2}	s_{S2}	0	D3
D2	x_{D2}	y_{D2}	$-d_{D2}$	e_{D2}	l_{D2}	s_{D2}	S3	0
S3	x_{S3}	y_{S3}	d_{S3}	e_{S3}	l_{S3}	s_{S3}	S1a	0
S3a	x_{S3}	y_{S3}	0	e_{S3}	l_{S3}	0	0	D2
S3b	x_{S3}	y_{S3}	0	e_{S3}	l_{S3}	0	0	D3a
D3	x_{D3}	y_{D3}	$-d_{D3}$	e_{D3}	l_{D3}	s_{D3}	S2	0
D3a	x_{D3}	y_{D3}	0	e_{D3}	l_{D3}	0	S3b	0

Table 42 - The Transformed Input Data Set

which establishes the pairwise precedence of

S1 - D1; S1a - S3; S3a - D2; S2 - D3; and S3b - D3a.

To convert to the transformed state, add an additional node if the original node has been used ensuring that the dummy node places no additional demand or service time on the problem.

Chapter 6

Areas for Further Research and Summary

6.1 Areas for Further Investigation

Several concerns arose during the course of this research. Each of these issues could use the algorithms developed during this research as a foundation to build upon, modify and improve. Despite the computational success of the NEW algorithms, most of these concerns are geared towards improving the efficiency and solution quality.

The efficiency and quality of algorithms can be greatly enhanced by using intelligent procedures for isolating effective candidate moves, rather than trying to evaluate every possible move in a current neighborhood of alternatives. This is particularly true when such a neighborhood is large and "expensive" to examine. Several areas relating to this issue can be studied.

The first area centers on identifying solutions and updating results from prior iterations. The two-level open hashing structure has proven to be an effective means for identifying and updating solutions efficiently for vehicle routing problems. Alternate measures can be investigated for saving and updating evaluations from previous iterations for the PDPTW which would reduce the overall computational effort. Savings earned can be devoted to higher level features of reactive tabu search.

The NEWc algorithm used a very basic candidate list strategy by accepting the first improving move encountered. Plus, each search neighborhood employed its own respective candidate lists to limit and expedite the search. Considerable benefits can result by employing alternate candidate list strategies. Knowledge of some fundamental candidate list approaches, for example, a sequential fan candidate list strategy or vocabulary building, could prove beneficial (refer to Glover 1995 and 1996).

The effective integration of candidate list strategies with the reactive tabu search memory designs will facilitate functions to be performed by the candidate lists. This applies especially to the use of frequency based memory. Recency based memory generates the history of the search and alters the tabu criteria based on the quality of the search path. Frequency based memory - which itself takes different forms in intensification and diversification phases - can not only have a dramatic impact on the performance of the search in general, but also can often yield gains in the design of candidate list procedures. Intensification strategies are based on recording and exploiting elite solutions. Two forms of intensification were employed in the algorithms used in this research. The first intensification strategy used was the within route insertion (WRI) neighborhood search strategy. WRI is invoked after *each* iteration of single pairwise-POS insertion (SPI). The SPI neighborhood search scheme has the greatest potential

for reducing the total travel time for the tour. The other intensification strategy employed returned the search to an elite solution at a fixed point in the search if an attractor basin had not been encountered. The NEW algorithms routinely found its best results early in the search. It could prove more beneficial to restart the search at an elite solution after so many stagnant iterations instead of waiting until reaching a fixed point. Other intensification approaches exist and may be incorporated into this research, for example, recovering elite solutions in some order. Other intensification approaches are outlined in Glover (1996, 43-45).

Diversification approaches depend on a purposeful blend of memory and strategy. Both recency and frequency based memory are important for diversification. Diversification is invoked in this research when too many solutions are visited within a certain span of iterations. The swap pairs neighborhood search scheme is used to catapult the search into another region of the solution space. The number of successive iterations of the swap pairs neighborhood used is based on the problem size and the number of routes within the tour. Other, more effective diversification approaches may consist of making a number of random moves proportional to a moving average of the cycle length.

The hierarchical approach, the multineighborhood strategic search methodology, is one method of employing the three neighborhood search strategies used in this research. Other neighborhood strategies may be

incorporated or new strategies to oscillate between the search neighborhoods invoked. For example, the swap pairs search neighborhood is executed if diversification is required or the SPI search neighborhood is locked out of any moves due to the presence of tight time windows. The swap pairs search neighborhood may be systematically implemented based on the average time window length (*atwl*) and the frequency memory strategy. When the *atwl* is small, the time windows are tight. Systematically calling the swap pairs neighborhood search routine will provide minor alterations to the makeup of the tour and assist in producing a quality search path. If the *atwl* is large, the swap pairs search neighborhood does not need to be called due to the large number of feasible solutions. The swap pairs neighborhood search routine would only be called when diversification is needed. However, if too many repeat solutions are being revisited too quickly, the swap pairs search neighborhood could be called to attempt to alter the search trajectory. This could help overcome the attraction of the chaotic attraction basin.

The final area of concern is to seek methods of expediting the SPI search process. This $O(n^3)$ neighborhood search routine is vital to determining the number of routes in the tour. The strength of SPI also resides in its ability to investigate potentially workable infeasible solutions. Improved coding efficiency would have to balance speed with SPI's ability to examine infeasible tours.

6.2 Extensions to this Research

The previous section discusses areas of concern that could be investigated for solving the PDPTW. This section discusses new areas for research that the NEW algorithm may be used as a basis for attack and presents further vehicle routing problems for investigation.

Solomon's VRPTW benchmark data sets were exploited for use with precedence constrained routing problems with time windows (PCRPTW) because no defined data sets existed for the PCRPTW or the PDPTW. A more extensive benchmark set of data needs to be generated, first for testing the PDPTW. Minor modifications to the PDPTW data sets can be made for testing the PCRPTW. Solomon's benchmark data sets do not bring vehicle capacity constraints into consideration. Therefore, the new problems generated should provide for all combinations of loose and tight capacity and time window constraints. Also, the problems should address a wider variety of customers, with vehicle restrictions, and number of required and available vehicles. After the problems are generated, the NEW algorithms could be used to test the problems' feasibility and create a set of best known solutions and tours.

The next primary area for potential extension is developing procedures to attack more general PDPTWs. An example of constraints to be included are route length restrictions which are not contingent upon time. Other

generalizations would definitely include non-homogeneous vehicles, vehicles with both weight and volume restrictions, multiple depot problems and alternate and hierarchical objective functions. For example, there are many possible characteristics which cause vehicles to be different. Vehicles could be segregated into a number of different vehicle categories. In this research, the $type_i$ variable was used the classify the node as either a customer node, type 1, or a vehicle node, type 2 (refer to section 3.4). This definition can easily be extended with $type_i = 1$ still being the customer node and $type_i = t$ for $t = 2$ to $m+1$ allowing for m different types of vehicles. The time it takes the vehicles to travel between customers may vary. A unique time-distance matrix would have to be generated for each vehicle class. The travel time from customer i to customer j on a vehicle in class k would then be $t_{i,j,k}$. This value would be used to compute appropriate move values and schedule parameters. Vehicle capacities, by weight and volume, may be input as an array for each vehicle type. This array could be referenced by either the vehicle identification number, Veh_i, or the vehicle type, $type_i$, as discussed earlier. This array would be passed to the function for computing penalties without significant changes in the computer code.

This research investigated a static or advanced request problem where the requirements are completely specified beforehand. This situation corresponds to routine support required for a system or a physical system which requires

customers to request service long enough in advance so that vehicles routes are completely determined before departing the depot. No further requests for service are accepted after the vehicles are dispatched. A dynamic problem, on the other hand, allows customers to request service after the vehicles are dispatched. The dynamic problem presents a more realistic approach to picking up and delivering supplies. Many requests are not known a priori. Rush orders are placed with customers all the time. Knowing the *slack* or waiting time inherent in all routes coupled with the average time window length (*atwl*) will aid the dispatcher in assigning the new request to a route. If the *atwl* is small, the time windows are tight for a higher percentage of customers being serviced. In general, the potential exists for drivers to spend a better portion of their schedule waiting to pickup the supplies and/or make the requisite delivery. A subroutine could be prepared to determine if the request could be handled by the available slack time in a route. An alternate subroutine would have to be developed if the *atwl* is large. If the *atwl* is large, the time windows are not that constraining for the customers being serviced. In general, the amount of slack time available is not as great. However, there exists greater flexibility to meet the demands of the customer due to the larger time windows. Thus, the subroutine could evaluate each route separately to determine if enough flexibility is available in the remaining schedule of a route to add the request. This dispatcher could then

forward the request to the driver on the route that minimizes the overall travel time added to the route.

The final area to explore is vehicle routing problems that allows a vehicle multiple routes during a single planning horizon. If the planning horizon is long and the number of vehicles available small, it could prove beneficial to allow the vehicles to leave and return to the depot several times during the planning horizon. Depot/vehicle nodes could be dynamically assigned to reflect the return of the vehicle to the depot, an appropriate amount of waiting/service time before the vehicle is permitted to leave for pickups and deliveries. A possible methodology for attacking this type of problem would be to initially divide up the planning horizon into segments representing the maximum number of routes a vehicle can complete within a planning horizon. The algorithm could be used to generate routes that satisfy this initial configuration. The next "phase" would find moves that decrease the overall travel time. Vehicle node early arrival and late departure times would be adjusted accordingly. Moves could be first considered between routes for the same vehicle and then between vehicles.

6.3 Major Contributions of this Research

This research makes several major contributions to the understanding and extension of knowledge in solving precedence constrained routing problems with time windows. This research also attempted to incorporate the concerns of

Glover (1996) and Battiti (1995) in employing (reactive) tabu search methodologies in solving vehicle routing problems. This research shows how to exploit dominant constraints to motivate the type of neighborhood search strategies utilized. Once identifying applicable search strategies, this research employs a multineighborhood strategic search methodology showing how to integrate the search neighborhoods into an effective hierarchical search scheme. Additionally, this research shows how to effectively adapt reactive tabu search methodologies, coupled with the multineighborhood strategic search methodology, to solve PDPTW.

The algorithms generated for this research successfully solved the PDPTW. The Solomon benchmark VRPTW data sets were modified to conform to the PDPTW data structure required, and, in the process, creating PDPTW benchmark data based on Solomon's benchmark data sets.

Several variants of the PCRPTW were presented - single supplier supporting several delivery locations, several suppliers supporting a single delivery location and serial precedence. This research showed how to transform these generalized precedent variants into the PDPTW.

This research identified difficulties in employing reactive tabu search. Simple confinements of the search trajectory can be cycles, an endless repetition of a sequence of solutions during the search. However, confinements can be

more complex trajectories with no clear periodicity but restricts the search to a limited portion of the solution space. This is what Battiti (1995) identifies as a chaotic attractor basin. The search does not want to expend too much effort attempting to overcome an attractor basin. The search wants to be identifying new solutions in an attempt to find optimal and infeasible tours that may further reduce the total travel time. The research discussed how to identify being caught in an attractor basin and illustrated one method for escaping the effects of the attractor basin. Successive iterations of the swap pairs search neighborhood were performed to alter the makeup of the routes and move the search in other regions of the solution space.

The algorithms employed traverse infeasible regions of the solution space in order to discover excellent solutions which might not otherwise be connected to the starting solution if the search neighborhoods were restricted to feasible solutions. In fact, the exploration of infeasible solutions is encouraged. The purpose of the search process is not only to find a solution that satisfies all constraints, but also to find infeasible tours that are viable and may further reduce the tour's total travel time.

6.4 Summary

This investigation has presented an effective reactive tabu search approach for solving precedence constrained routing problems with time

windows. This effort primarily focused on solving the PDPTW and show how to transform the more complicated PCRPTW into a solvable PDPTW.

The computational results show the procedures to be efficient, producing solutions in an order of magnitude faster than the best known optimal approaches and heuristic approaches for the VRPTW. The solutions generated often match the optimal/best solution, especially when the optimal schedule for the VRPTW was known. The search process identified infeasible tours that lowered the objective value. These tours further decrease the total travel time by slightly violating time window constraints. Some of these time windows violations can best be explained to the customer as the vehicle "got caught in traffic". Significant time windows violations can be negotiated with the customer. Thus, both the customer and the distributor gain additional savings.

Despite the broad spectrum of VRPs that have been identified and solved during this research, data obtained from the Defense Logistics Agency (DLA) indicated future needs for research as outlined in this chapter.

APPENDIX A

Data Sets Examined

The data sets examined for comparison in this research are the 25-, 50- and 100-customer problem instances that the team of Desrochers, Desrosiers and Solomon (1992) were able to optimally solve using their optimal algorithm. The optimal algorithm produces optimum solutions for all twenty-nine, 25-customer problems, fourteen of the 50-customer problems and seven of the 100-customer problems. A synopsis of their procedure is outlined below.

The linear programming (LP) relaxation of the set partitioning formulation of the VRPTW is solved by column generation. Feasible columns are added as needed by solving a shortest path problem with time windows and capacity constraints using dynamic programming. The LP solution obtained generally provides an excellent lower bound that is used in a branch-and-bound algorithm to solve the integer set partitioning formulation. Desrochers, Desrosiers and Solomon treat the case where if a vehicle arrives prior to the early time window boundary at a location, the vehicle will wait. Late time window boundaries cannot be violated. The "homogeneous" fleet size is determined simultaneously with the best sets of routes and schedules rather than being fixed a priori. Each customer is serviced exactly once.

In order to use their pulling algorithm, the authors provide a definition for lexicographic ordering based on the capacity needed at the node and the time to get to the node. The authors use this pulling algorithm to generate negative marginal cost columns. When no more of these columns can be generated, the simplex algorithm provides the optimal solution of the linear relaxation of the set covering type model. If the solution is integer, it is also optimal. Regardless of the integrality of the solution, a branch-and-bound strategy is used to solve the integer set partitioning formulation.

The authors use three of the six benchmark problem sets generated by Solomon; r1, c1 and rc1. These problems have a short scheduling horizon and vehicle capacity is not a serious consideration. Computational results are provided for all variants of the problems. It proved capable of solving the following problems exactly:

25-customers: all twenty-nine problem instances.

Clustered: c101 - c109,
Radially dispersed: r101 - r112, and
Mixed: rc101 - rc108.

50-customers: fourteen of the twenty-nine problem instances.

Clustered: c101, c102, c103, c105, c106, c107 and c108.
Radially dispersed: r101, r102, r103, r105, r106, r107 and r110.

100-customers: seven of the twenty-nine problem instances.

Clustered: c101, c102, c106, c107 and c108.
Radially dispersed: r101 and r102.

Kohl (1995) purports to solve 70 of the 87 benchmark problems to optimality. Knowing the number of routes in the tour for of the optimal solution enabled the scope of the problems investigated to be expanded. Carlton's RTS algorithm was used to try to identify the optimal solution or a solution with the same number of routes.

Kohl's method exploits Lagrangian relaxation of the constraint set requiring that all customers must be serviced. Kohl decomposes the VRPTW into a master problem and a shortest path problem, much like the Dantzig-Wolfe decomposition, but the way the dual variables or Lagrangian multipliers are optimized is new. The master problem finds the optimal Lagrangian multipliers by using a method exploiting the benefits of subgradient methods as well as a bundle method. The subproblem is a shortest path problem with time window and capacity constraints (SPPTW). The method was implemented on the Solomon benchmark problems of size up to 100-customers. The algorithm proved to very competitive and succeeded in solving several previously unsolved problems. The problems optimally solved are provided below.

25-customers: all twenty-nine problem instances.

50-customers: twenty-seven of the twenty-nine problem instances.

Clustered: all nine problems, c101 - c109.
Radially dispersed: ten of twelve problems, except r108 and r112.
Mixed: all eight problems, rc101 - rc108.

100-customers: fourteen of the twenty-nine problem instances.

Clustered: all nine problems, c101 - c109.
Radially dispersed: r101, r102 and r105.
Mixed: rc101 and rc105.

APPENDIX B

Example of the Modified Data Structure

The first table in this appendix presents the data structure for Solomon's radially dispersed, r110, 25-customer VRPTW. The first two columns present the (x, y) coordinate location of the customer. The third column refers to the demand of the customer. For the VRPTW this refers to either all customers receiving goods or all customers providing goods for whatever measure is being used, i.e., by weight or volume. The fourth and fifth columns are the time windows for the customers, reflecting when the customer wants the product delivered or picked up. The final column tells how long it will take to load or unload the supplies at each location. The first row in the table provides the information for the depot node. All vehicle nodes are modeled after this depot node. The late arrival time for the depot provides the planning horizon.

Table B.2 shows the changes and additions that had to be made to convert the VRPTW to a PDPTW problem instance. The first six columns provide the same information with one notable change. The demand data has been changed to reflect whether the demand will be added to the total load of the vehicle (+) or removed from the total load of the vehicle (-). Plus, the demand values have been altered to the *pairwise*-POS. The supplier has a positive demand *d* and the corresponding delivery location has the negative demand -*d*. The seventh

column identifies if there is a predecessor node and who that predecessor node is. The eighth column identifies if there is a successor node and who that successor node is. If the predecessor column contains a nonzero entry, the corresponding entry in the eighth column will be zero. This situation indicates that the customer is a successor or delivery node. The final observation to make occurs in row 10. The service time in row 10 has been zeroed out. This customer, node 9, was the last customer in a route with an *odd* number of customers for the VRPTW. The last row in table B.2 is a dummy node modeled after node 9. The dummy node contains the service time for node 9. This dummy node serves as the delivery node and successor to node 9 establishing the node pair (9, 26) as a *pairwise*-POS.

The optimal tour for r110, which satisfies the imposed precedence and coupling constraints, is provided in table B.3. The optimal tour requires 5 vehicles with a total travel time of 694.1 units. The vehicle nodes delineating the ending of the routes are highlighted. The vehicle nodes are modeled after the depot node and reflect when the vehicle returns to the depot. The largest return time, 202.6 for vehicle 28, is the tour makespan.

Node	x[i]	y[i]	demand	early arrival time	late arrival time	service time
Depot	35	35	0	0	230	0
1	41	49	10	130	201	10
2	35	17	7	20	89	10
3	55	45	13	106	135	10
4	55	20	19	71	195	10
5	15	30	26	20	107	10
6	25	30	3	54	153	10
7	20	50	5	66	105	10
8	10	43	9	61	138	10
9	55	60	16	53	150	10
10	30	60	16	101	156	10
11	20	65	12	33	152	10
12	50	35	19	38	97	10
13	30	25	23	70	208	10
14	15	10	20	32	137	10
15	30	5	8	30	154	10
16	10	20	19	54	105	10
17	5	30	2	51	189	10
18	20	40	12	77	106	10
19	15	60	17	53	108	10
20	45	65	9	109	152	10
21	45	20	11	37	96	10
22	45	10	18	59	144	10
23	55	5	29	36	155	10
24	65	35	3	118	190	10
25	65	20	6	47	186	10

Table B.1 - Original Solomon Data for 25-customer r110.

Node	x[i]	y[i]	demand + pickup - deliver	early arrival time	late arrival time	service time	pred[i]	succ[i]
Depot	35	35	0	0	230	0	0	0
1	41	49	-19	130	201	10	20	0
2	35	17	17	20	89	10	0	15
3	55	45	-19	106	135	10	12	0
4	55	20	19	71	195	10	0	24
5	15	30	-12	20	107	10	18	0
6	25	30	23	54	153	10	0	13
7	20	50	15	66	105	10	0	19
8	10	43	-22	61	138	10	17	0
9	55	60	16	53	150	0	0	26
10	30	60	-16	101	156	10	11	0
11	20	65	16	33	152	10	0	10
12	50	35	19	38	97	10	0	3
13	30	25	-23	70	208	10	6	0
14	15	10	20	32	137	10	0	16
15	30	5	-17	30	154	10	2	0
16	10	20	-20	54	105	10	14	0
17	5	30	22	51	189	10	0	8
18	20	40	12	77	106	10	0	5
19	15	60	-15	53	108	10	7	0
20	45	65	19	109	152	10	0	1
21	45	20	18	37	96	10	0	22
22	45	10	-18	59	144	10	21	0
23	55	5	29	36	155	10	0	25
24	65	35	-19	118	190	10	4	0
25	65	20	-29	47	186	10	23	0
26	55	60	-16	53	150	10	9	0

Table B.2 - Modified Data Structure for PDPTW - nr110.

node i	arr[i]	dep[i]	l[i]
21	18	37	96
22	57	59	144
23	80.1	80.1	155
4	105.1	105.1	195
25	125.1	125.1	165
24	150.1	150.1	190
27	190.1	0	230
7	21.2	66	86.9
19	87.1	87.1	108
11	104.1	104.1	134.9
10	125.2	125.2	156
20	151	151	152
1	177.4	177.4	201
28	202.6	0	230
12	15	38	97
3	59.1	106	135
9	131	131	150
26	131	131	150
29	173	0	230
2	18	20	89
15	43	43	154
14	68.8	68.8	83.9
16	89.9	89.9	105
17	111	111	114.1
8	134.9	134.9	138
30	171.1	0	230
18	15.8	77	85.9
5	98.1	98.1	107
6	118.1	118.1	153
13	135.1	135.1	208
31	156.2	0	230

Table B.3 - Optimal Tour for nr110

APPENDIX C

The Code for the PDPTW Algorithm

```
/* NEW.C for PC */
/* Updated 10/12/97. This uses BUILD_TOUR to generate a feasible initial tour. It accounts for
getting caught in a chaotic attractor basin. If it gets caught in an attractor basin, it diversifies
using swap_pairs. If it gets caught twice, it restarts the whole search process at the best tour and
reinitializes the search parameters.*/

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <limits.h>
#include <string.h>
#include <time.h>

#define MAXPBM 20    /* The longest problem name allowed.*/
#define max(a, b) ((a>b) ? (a) : (b))
#define min(a, b) ((a<b) ? (a) : (b))

/* Function prototypes. */
void main(void);
void input_fn(char *);
void input_vpen(double *pvpen);
struct hashlist *lookfor(long, unsigned long, long, long, int, long);
void notfound(int *tl, int *ss, float mavw, FILE *ofp);
void found(struct hashlist *ptr, int *tl, int *ss, float *ma, int k, FILE *ofp);
void makespan (int, struct node *, long *, int *);
int count_vehs(int, struct node *);
void print_tour(struct node *t, int n, FILE *ofp);
void print_sched(struct node *t, int n, FILE *ofp);
void keep_bfs(int, struct node *, long, long, struct node *, long *, long *, int *, int,
   clock_t,double *);
void twredol(int nnodes, struct node *tour, int **time, FILE *ofp);
void print_tws(int n, struct node *t, FILE *ofp);
void input_pbm (int nc, int nv, int g, struct node *t, int **ti, FILE *ifp, FILE *ofp);
void convxy(int nc, int nv, int gamma, float *x, float *y, float *s, int *pr, int *su, int **ti, FILE
   *ofp);
float *t_search(FILE *ifp,  FILE *ofp, int niters, double TWPEN, int g, double VPEN, int CAP,
   char INFILE);
int move_delta (int, int, int, struct node *, struct node *, int **);
long tour_sched (int, int, struct node *, int **);
void comp_parpens (struct penalty *, int,struct node *, int);
void swap_node (int, int, struct node *);
struct node * insert (int, struct node *, int *, int, int);
```

```
long sum_wait(int, struct node *);
void iinsert(int, struct node *, int, int);
void build_tour(struct node *, int **, int, int, int);

void input_vpen(double *pvpen)
{
  int i;     /* Index */
  char c, d;  /* Characters to read y or n to change vpen */
  double p;  /* Dummy value for vpen for input */
  printf("The current overload penalty is: %6.1f.\n", *pvpen);
  printf("Do you want to change the value? <Y, N>\t");
  c = getchar();  /* This captures any '\n' characters leftover */

  for (i=0; (c = getchar()) != '\n'; ++i)
  d = c;
  while (d != 'N' && d != 'n') {
    printf("\nInput the new overload penalty.\n");
    scanf("%lf", &p);
    c = getchar();  /* This captures any '\n' characters leftover */
    *pvpen = p;
    printf("The current overload penalty is: %6.1f.\n", *pvpen);
    printf("Do you want to change the value? <Y, N>\t");
    for (i=0; (c = getchar()) != '\n'; ++i) d = c;
  } /* end while */
}/* end of the input vpen function */

/* Define values to examine in the solution structure */
#define HTSIZE 1009     /* The dimension of the hashing table */
/* REACTIVE SEARCH PARAMETER INITIALIZATIONS */
#define INCREASE  1.5     /* Should guarantee to increase tabu length by at least one */
#define DECREASE  0.95  /* Allows for any function to be inserted */
#define CYMAX  50         /* Allows for any function to be inserted */
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))

typedef struct node { /* The data structure for a vehicle or "task" node.*/
/* Input data */
   int id;        /* The node number.*/
   int e;         /* The early time window.*/
   int l;         /* The late time window.*/
   int qty;       /* The customer demand or 0.*/
   int type;      /* The node type: 1 => customer, 2 => vehicle.*/
   int pred;      /* The (vector) of node predecessors */
   int succ;      /* The (vector) of node successors */
   int vehicle;  /* Records the vehicle the customer node is on */
/* Schedule Variables */
   long arr;      /* The arrival time at the node.*/
   long dep;      /* The departure time from the node.*/
```

```
   long wait;    /* The waiting time at the node.*/
   long load;    /* The amount loaded on the vehicle at the node.*/
} NODE;

/* The structure for the penalty terms */
typedef struct penalty { /* The data structure for the penalty terms.*/
   long tw;              /* The total amount of TW violation for a tour.*/
   long ld;              /* The total amount of overload for a tour.*/
} PENALTY;
/* Again, precedence and coupling constraints are the dominant concerns.  Each tour created will
have these constraints satisfied. */

/* The hashing structure */
struct hashlist {
   unsigned long thval;    /* The tour hashing value.*/
   long cost;              /* The cost of the tour.*/
   long tvltime;           /* The tour travel time.*/
   long twpen;             /* The tour's TW penalty value.*/
   long loadpen;           /* The tour's overload penalty value.*/
   int lastfound;          /* The iteration on which the tour was last visited.*/
   int repeat;             /* To record how many repeat visits were made to the tour */
   struct hashlist *next;  /* Pointer to the next item in the list.*/
} HASHLIST;

/* The hashing table is a vector of pointers to hashlist structures. */
   struct hashlist *hashtbl[HTSIZE];

void twredol(int nnodes, struct node *tour, int **time, FILE *ofp)
/*  Input parameters; Tour and schedule parameters  */
   int nnodes;          /* the total number of modelled nodes.*/
   int **time;          /* the time (distance) matrix between nodes.*/
   struct node *tour;   /* defines a tour as a vector of node structs.*/
   FILE *ofp;           /* the pointer to the problem output file.*/
{
   register int i;      /* index for counting */

/*              BEGIN THE PDPTW REDUCTION FUNCTION                    */

for (i=1; i <= nnodes; ++i)  {
   if (tour[i].pred !=0) tour[i].l = min(tour[i].l,tour[0].l - time[i][0]);
}
for (i=1; i <= nnodes; ++i) {
   if (tour[i].succ !=0) tour[i].l = min(tour[i].l,tour[tour[i].succ].l - time[i][tour[i].succ]);
}
for (i=1; i <= nnodes; ++i) {
   if (tour[i].succ !=0) tour[i].e = max(tour[i].e,tour[0].e + time[0][i]);
}
```

```
for (i=1; i <= nnodes; ++i) {
   if (tour[i].pred !=0) tour[i].e = max(tour[i].e,tour[tour[i].pred].e + time[tour[i].pred][i]);
}
   return;
} /* This ends the PDPTW reduction function. */

/************* This function prints the time windows *******************/
void print_tws(int n, struct node *t, FILE *ofp)
   struct node *t;      /* The tour to be printed.*/
   int n;               /* The number of nodes in the tour.*/
   FILE *ofp;           /* The pointer to the problem output file.*/
   int i;               /* Index */
{
   fprintf(ofp,"\nThe time windows are: \n\n");
   fprintf(ofp,"%s%6s %7s %7s \n", " ", "node i", "e[i]", "l[i]");

   for (i=0; i<n; ++i) {
      fprintf(ofp,"%7d %7d %7d\n", t[i].id, t[i].e, t[i].l);
   } /* end for */
   fprintf(ofp,"\n");
} /*end of time window printing function.*/

/* This program converts an input file of (x,y) integer coordinates to a distance matrix ti(i,j) for
an n node TSPTW, where n = number of nodes including the depot the program outputs the
matrix of INTEGERS ti(i,j) to a designated time (distance) file for the Solomon set of problems.
The data is scaled by a factor of 10 and truncated.  */

#include <math.h>
#define FACTOR 10.0  /* Multiplies TWs and t(i,j) & s(i)'s to increase accuracy, yet use integer
   computations.*/
#define TRAVEL 1.0

void input_pbm(int nc, int nv, int g, struct node *t, int **ti, FILE *ifp, FILE*ofp)
{
   float *x, *y;        /* The x, y coordinates of the customers and the depot.*/
   float *ea, *la;      /* The input vectors for the early and late TWs, respectively.*/
   float *s;            /* Service time.*/
   int *d;              /* Demand of customer i */
   int *pr;             /* list of suppliers or customers that must precede this location */
   int *su;             /* list of suppliers or deliveries that must succede this location */
   int i;               /* Index.*/
   x = (float *)calloc(nc, sizeof(float));
   y = (float *)calloc(nc, sizeof(float));
   ea = (float *)calloc(nc, sizeof(float));
   la = (float *)calloc(nc, sizeof(float));
   d = (int *)calloc(nc, sizeof(int));
   s = (float *)calloc(nc, sizeof(float));
```

```
  pr = (int *)calloc(nc, sizeof(int));
  su = (int *)calloc(nc, sizeof(int));

/* Input Depot and Customer data from the input file.*/

for (i=0; i < nc; ++i)  {
  t[i].id = i;
  t[i].type = 1;
    fscanf(ifp,"%f", &x[i]);
    fscanf(ifp,"%f", &y[i]);
    fscanf(ifp,"%d", &d[i]);
    fscanf(ifp,"%f", &ea[i]);
    fscanf(ifp,"%f", &la[i]);
    fscanf(ifp,"%f", &s[i]);
    fscanf(ifp,"%d", &pr[i]);
    fscanf(ifp,"%d", &su[i]);
  t[i].e = (int) (FACTOR)*ea[i];
  t[i].l = (int) (FACTOR)*la[i];
  t[i].qty = d[i];          /* this is + for a supplier; - for a delivery */
  t[i].pred = pr[i];
  t[i].succ = su[i];
  t[i].vehicle = nc+nv; /* initialize for use after build_tour */
}/* end for */
  t[0].type = 2;          /* reset the 0-depot to the correct type.*/
/* Initialize the remaining vehicle nodes to be the same as the depot node "Homogeneous" */
for (i = nc; i < nc+nv; ++i) {
  t[i].id = i;
  t[i].e = t[0].e;
  t[i].l = t[0].l;
  t[i].type = 2;
  t[i].qty = t[0].qty;
  t[i].pred = t[0].pred;
  t[i].succ = t[0].succ;
  t[i].vehicle = i;
} /* end for */
  free(ea);
  free(la);
  free(d);

  convxy(nc, nv, g, x, y, s, pr, su, ti, ofp);

  free(x);
  free(y);
  free(s);
  free(pr);
  free(su);
  return;
} /* end input_pbm function */
```

```
void convxy(int nc, int nv, int gamma, float *x, float *y, float *s, int *pr, int *su,
   int **ti, FILE *ofp)
/* int nc;          The number of customers incl the depot.
   int nv;          The number of vehicles.
   int gamma;       The penalty value for using an add'l veh.
   float *x, *y;    The x,y coordinates of nodes.
   float *s;        Service time vector.
   int *pr, *su;    Node predecessor and successor arrays.
   int **ti;        The resulting time (distance) matrix.
   FILE *ofp;       The pointer to the problem output file.*/
{
   int i,j;              /* Indices. */
   double dx, dy;        /* The difference between the respective x and y coordinates.*/
   double power();       /* The power function prototype. */
   int nn;               /* Computes nnodes.*/

   nn = nc + nv;
   for (i=0; i<nn; ++i)   /* Initialize the distance matrix. */
     for (j=0; j<nn; ++j) ti[i][j] = 15000;

/* Account for 0's from depot-supply; delivery-depot */
   for (j=1; j<nc; ++j) {
     dx = x[0] - x[j];
     dy = y[0] - y[j];
     if (pr[j] == 0)  ti[0][j]= (int) (TRAVEL*FACTOR*(sqrt(dx*dx + dy*dy)));
/* Assume that the average rate of travel = 30 mph */
     else ti[j][0]= (int) (TRAVEL*FACTOR*(sqrt(dx*dx + dy*dy)));
   } /* end for j */

for (i=1; i<nc; ++i) /* Computation. */
   for (j=i+1; j<nc; ++j) {
     dx = x[i] - x[j];
     dy = y[i] - y[j];
     if (j!=su[i]){
       ti[i][j] = (int) (TRAVEL*FACTOR*(sqrt(dx*dx + dy*dy)));
       ti[j][i] = ti[i][j]; /* assumes time-distance symmetry */
     } /* end if */
     else ti[i][j] = (int) (TRAVEL*FACTOR*(sqrt(dx*dx + dy*dy)));
   } /* end for j */
   for (i=1; i< nc; ++i)
     for (j = nc; j < nc+nv; ++j) {
     if (ti[0][i] != 15000) {
       ti[j][i]=ti[0][i];
         ti[i][j]=15000;
       } /* end if */
       else {
       ti[j][i]=15000;
```

```
        ti[i][j]=ti[i][0];
      } /* end else */
    } /* end for */

/* Add scaled service time to each distance vector.*/
  for (i=0; i< nc; ++i)
  for(j=0; j<nc+nv; ++j)
    if (ti[i][j] != 15000) ti[i][j] +=(int) FACTOR*s[i];
/*divide s[i] by two for "half s"*/

  for (i=nc; i< nc+nv; ++i)
  for(j=0; j< nc+nv; ++j)
    ti[i][j] += (int) FACTOR*s[0];
/*divide s[0] by two for "half s"*/
/* Complete the matrix by inputting the variable vehicle usage "cost."*/
  for (i = nc; i < nn-1; ++i)
  for (j= i+1; j < nn-1; ++j)
      ti[i][j] = ti[j][i] = gamma;
  for (i = nc; i < nn-1; ++i)
    ti[i][0] = ti[0][i] =ti[i][nn-1] = ti[nn-1][i] = gamma;
  return;
}

/* A function to print the current tour. */
void print_tour(struct node *t, int n, FILE *ofp)
{
  register int i;   /* Index */
  fprintf(ofp,"The tour is now:\n");
  for (i=0; i < n; ++i)
  fprintf(ofp,"%d ", t[i].id);
  fprintf(ofp,"\n\n");
  return;
} /* This is the end of the function. */

/* A function to print the schedule. */
void print_sched(struct node *t, int n, FILE *ofp)
{
  register int i;/* Index */
  fprintf(ofp,"\nThe resulting schedule is: \n\n");
  fprintf(ofp,"%s%6s %7s %7s %7s \n", " ", "node i", "arr[i]", "dep[i]", "l[i]");
  for (i= 0; i < n; ++i) {
    fprintf(ofp,"%7d %7.1f %7.1f %7.1f", t[i].id, (float) t[i].arr/(float)
      FACTOR, (float) t[i].dep/(float) FACTOR, (float) t[i].l/(float) FACTOR);
    if (t[i].arr > t[i].l) fprintf(ofp,"*\n");
    else fprintf(ofp,"\n");
  } /* end for */
  fprintf(ofp,"\n"); }/*end of schedule printing function.*/
```

```
}
/* This is the function that retains the feasible solution that has the shortest travel time and with
the shortest travel time also has the shortest completion time, as well from among all tours
"found" in the TS routine. This is a much better routine than previous ones, too.*/

void keep_bfs(n, t, tc, tt, bft, pbftc, pbftt, pbfin, k, st, pbfti)
   int n;            /* The number of nodes in the tour.*/
   NODE *t;          /* The structure of the current feasible tour.*/
   long tc;          /* The current tour cost.*/
   long tt;          /* The current tour travel time.*/
   NODE *bft;        /* The struct for the best feasible tour.*/
   long *pbftc;      /* The pointer to the best feasible tour cost.*/
   long *pbftt;      /* The pointer to the best feasible travel time.*/
   int *pbfin;       /* The pointer to iteration number when bfs was found.*/
   int k;            /* The current iteration number.*/
   clock_t st;       /* The time the algorithm started.*/
   double *pbfti;    /* Pointer to best feasible solution found time.*/
{
   int i;            /* Index.*/
   long travel;      /* The travel time for the current tour.*/
   clock_t solntime; /* The current time for the algorithm.*/
   travel = tt;
   solntime = clock();
   if (travel > *pbftt)  return;
   if (travel < *pbftt) {
      *pbfti = ((solntime - st)/(double) CLOCKS_PER_SEC);
      *pbftc = tc;
      *pbftt = travel;
      *pbfin = k;
      for (i = 0; i < n; ++i) bft[i] = t[i];
      return;
   }/*end if travel < *pbftt.*/

   if (travel == *pbftt  &&  tc < *pbftc)  {
      *pbfti = ((solntime - st)/(double) CLOCKS_PER_SEC);
      *pbftc = tc;
      *pbfin = k;
      for (i = 0; i < n; ++i) bft[i] = t[i];
      return;
   }/*end if tour cost is equal.*/
   return;
}/*end of keep_bfs function*/

/* This function is like the one above except it retains the feasible tour which uses the smallest
number of vehicles.*/

/* The following functions handling the open hasing structure */
```

/* This function looks for the current tour in the hashing structure; if the tour is found, the pointer is returned, if not, the tour is added to the structure and a NULL pointer is returned.*/

```
struct hashlist *lookfor(tc, thv, twp, lp, k, tt)
   long tc;                /* The tour cost of the tour.*/
   unsigned long thv;      /* The hashing value of the tour.*/
   long twp;               /* The TW infeasibility penalty for the tour.*/
   long lp;                /* The overload infeasibility penalty for the tour.*/
   int k;                  /* The iteration number corresponding to the tour.*/
   long tt;                /* The tour travel time.*/
{
   struct hashlist *hnp;   /* The pointer to the new hashing tour position.*/
   struct hashlist *npl;   /* The pointer to the last tour in the "tree."*/
   unsigned hv;            /* The hashvalue for the tour cost.*/
   int hold;

   hv = (unsigned) (tc % HTSIZE);
   npl = NULL;
/* Look for the tour in the structure.If found, return the pointer.*/
   for (hnp = hashtbl[hv]; hnp != NULL; hnp= hnp->next) {
     npl = hnp;
     if (hnp->twpen == twp && hnp->loadpen == lp && hnp->thval == thv) {
          hold = hnp->repeat;
          hnp->repeat = hold+1;
       return hnp;
     }
   }/* end for */
/* If not found, add the tour to the end of the structure.*/
   hnp = (struct hashlist *)malloc(sizeof(HASHLIST));
   if (npl == NULL) {
     hnp->next = hashtbl[hv];
     hashtbl[hv] = hnp;
   }
   else {
     hnp->next = npl->next;
     npl->next = hnp;
   }/* end else */

   hnp->twpen = twp;
   hnp->cost = tc;
   hnp->thval = thv;
   hnp->lastfound = k;
   hnp->repeat = 1;
   hnp->tvltime = tt;
   hnp->loadpen = lp;
   return NULL;
} /* end function lookfor */
```

```
/* This function updates the search parameters if the incumbent tour is not found in the hashing
structure.*/

void notfound(int *tl, int *ss, float mavw, FILE *ofp) {
   *ss += 1;
/* If the current tour is not in the hashing table, then decrease the value for the tabu_length only
if the number of steps since the last change is more than  the current moving average .*/
   if (*ss>mavw) {
      *tl = max( (int) (*tl)*DECREASE,5);
      *ss = 0;
   }/* end if */
} /* end of function notfound */

/* This function updates the search parameters if the incumbent tour is found in the hashing
structure.*/

void found(struct hashlist *ptr, int *tl, int *ss, float *ma, int k, FILE *ofp)
   struct hashlist *ptr;   /* The pointer to the tour that was found.*/
   int *tl;                /* The pointer to the tabu_length.*/
   int *ss;                /* The pointer to the ssltlch.*/
   float *ma;              /* The current value for the moving average.*/
   int k;                  /* The current iteration count.*/
   FILE *ofp;              /* The pointer to the output file.*/
{
   int cylength;           /* The length of the current cycle.*/
 /* This updates the parameters for the search and increases the tabu_length if the cycle length is
less than the max allowable cycle length (CYMAX).*/

   *ss += 1;
   cylength = k - (ptr->lastfound);
   ptr->lastfound = k;
   if (cylength < CYMAX) {
      *ma = .1*cylength + .9*(*ma);
      *tl = min(2000, (int) (*tl)*INCREASE);
      *ss = 0;
   } /* end if */
 } /* end of function found */

/* This routine finds the maximum tour completion time for the algorithm. It also finds the
number of vehicles used within the tour.  Any time the arrival time at a vehicle node is greater
than zero, a vehicle has been used; therefore, increase the number of vehicles used count and if
the arrival time is larger than any previous arrival time, it becomes the mission completion
(makespan).*/

void makespan(n, t, ct, vu)
   int n;              /* The number of modeled nodes in the tour.*/
   NODE *t;            /* The tour structure.*/
```

```
  long *ct;          /* The pointer to the completion time value.*/
  int *vu;           /* The pointer to the number of vehicles used.*/
{
  long zct = 0;      /* The initial value for ZcT.*/
  int nvu = 0;       /* The initial value for number vehicles used.*/
  NODE *np;          /* The index pointer to the tour structure.*/
  NODE *lastnp;      /* The index pointer to the last tour node.*/
  lastnp = &t[n-1];
  for (np = &t[1]; np <= lastnp; ++np) {
    if (np->type == 1) continue; /* Skip all customer nodes.*/
    if (np->arr > 0) {
      ++nvu;
      if (np->arr > zct) zct = np->arr;
    }/* end if arrival at vehicle node is greater than zero.*/
  }/* end for tour search.*/
  *vu = nvu;
  *ct = zct;
  return;
}/*end makespan function.*/

/* This procedure counts the number of vehicles used and returns that integer.*/

int count_vehs(n, t)
  int n;             /* The number of modeled nodes.*/
  NODE *t;           /* The tour for counting.*/
{
  int nvu = 0;       /* The number of vehicles used by the input tour.*/
  NODE *np;          /* An index for traversing the tour.*/
  NODE *lastnp;      /* The last node pointer in the tour.*/
  lastnp = &t[n-1];
  for (np = &t[1]; np <= lastnp; ++np) {
    if (np->type == 1) continue; /* Skip all customer nodes.*/
    if (np->arr > 0)  ++nvu;
  }/*end for tour search.*/
  return nvu;
}/* End of the count vehicles function.*/
/ * The Tabu Search Routine */

floatt *t_search(FILE *ifp,  FILE *ofp, int niters, double TWPEN, int g, double
  VPEN, int CAP, char INFILE)

  /* FILE *ifp;        The pointer to the problem input file.*/
  /* FILE *ofp;        The pointer to the problem output file.*/
  /* int niters;       The number of iterations that the algorithm is allowed to run.*/
  /* double TWPEN;     The penalty for TW infeasibility.*/
  /* int g;            The penalty for additonal vehicle.*/
  /* double VPEN;      The penalty for overload infeasibility.*/
```

```
  /* int CAP;            The capacity for the vehicles in this pbm.*/
{
 /* VARIABLES  */
/* Program parameters:  */
   register int i, j, j2, d, d2;     /* Indices for counting */
   int k;                            /* k = iteration number */
   int temp1;                        /* Temporary values used by the SWAP macro.*/
   struct node temp2;
/* Input parameters:  */
   int numveh;
   int nnodes;          /* The total number of modelled nodes = numcust + numveh.*/
   int **time;          /* The time (distance) matrix between nodes.*/
   int tabu_length;     /* The number of iterations that a move is declared tabu.*/
   int depth;           /* The depth of the insertion moves.*/
/* Tour and schedule parameters:  */
   struct node *tour;  /* Defines a tour as a vector of node structs.*/
/* Tabu Search parameters:  */
   int **tabu_list;          /* The tabu list matrix: does not allow node i to reverse a prior swap
                                move with node j.*/
   struct node *oldtour;     /* A pointer to a previous candidate tour.*/
   struct node *oldtour2;    /* A pointer to a previous candidate tour.*/
   struct node *initial_tour;
   long tour_length;
/* The length of the neighboring tour = arrival at the depot (n) for the move.*/
   long move_val, move_val2;
/* The difference betweeen the candidate tour cost and the incumbent tour cost.*/
   long tour_cost;           /*The actual total cost of the move= tour_length + penalty.*/
   long d_best;              /*The smallest swap cost found among all neighbors.*/
   int ch_i, ch_d, ch_spi;   /*The tour positions that correspond to the best neighboring move
                               that is not tabu.*/
   long pen_tt;              /*The travel time + penalties for the incumbent tour.*/
   long compl_time;          /*The completion time for the best tour.*/
   long totwait;             /*The total waiting time for the given tour.*/
   int feas_i;          /* The index for the selected predecessor.*/
   int feas_d;          /* The depth index for the successor move.*/
   int feas_spi;        /* where predecessor is inserted before */
   long d_bestf;        /* The smallest swap cost found among all feasible neighbors.*/
   long best_tt;        /* The best travel time found so far, it includes the infeasibility penalties.*/
   long pen_cost;       /* The total penalty cost of the incumbent  tour.*/
   long tvl;            /* The travel time for the incumbent tour.*/
 /* Output parameters:  */
   struct node *best_tour;   /* The tour that has the smallest overall cost.*/
   long best_cost, iter_no;/* The cost of the best tour and iteration on which it was discovered.*/
   int numfeas;         /* The number of feasible tours found during the search, including
                           uplicates.*/
   long bt_pen_cost;         /* The penalty cost of the best tour found.*/
   float soln[15];           /* The solution values.*/
```

```
  struct node *best_ftour;      /* A node structure for the best feasible tour.*/
  long bftour_cost;             /* The value of the best feasible tour cost.*/
  long bftvl_time;              /* The smallest travel time for the best ftour.*/
  int bfiter_no;                /* The iter no when the bftour was found.*/
  int numdiff;                  /* The number of different solutions found.*/
  long feas_compl;              /* The completion time of best feasible tour.*/
  int num_veh_used;             /* The number vehs used in the best tour.*/
  int num_feas_veh;             /* The number vehs used in the best feas tour.*/
  FILE *fpinit;                 /* The file pointer to "initial.out".*/

/* Variables for the PDPTW Application.*/
  int numcust;          /* The number of customers in the problem.includes the depot.*/
  long load_cost;           /* The penalized overload = VPEN*load_penalty*/
  long tw_cost;         /* The penalized time window lateness = TWPEN*time_penalty*/
  long tot_penalty;     /* The total, unpenalized infeasibility of the incumbent tour =
                           time_penalty + load_penalty.*/
  long tot_nbrpen;      /* The total uncosted penalties of a nbr tour.*/
  int vehicle_id = 0;   /* Store id of current vehicle number in tour */
  int *locx, *loctemp, *loctemp2;
/* locator arrays */
  int *loc_initial;
  int current_vehicle;
  int same, restart = 0;
  int q = 0, count = 0;
  int first = 0, second = 0, third = 0;
  int wricount = 0, spicount = 0, swapcount = 0;
  int current = 0, next = 0, revisit = 0;

/* These are the needed structure variables.*/
  PENALTY tourpen; /* The penalty structure for the current tour.*/
  PENALTY nbrpen;  /* The penalty structure for a neighbor tour.*/

/* Timing variables: */
  clock_t start, stop;   /* Start and stop times of the TS algorithm */
  double duration;       /* duration = stop-start => total TS algoritm time */
  clock_t soln_time;     /* The time the current solution is found.*/
  clock_t init_end;      /* The time the initial solution is computed.*/
  double best_time;        /* The time the best solution is found.*/
  double bestf_time;     /* The time the best feasible soln is found.*/
  double init_time;      /* The computation time for the initial tour.*/
/* Hashing and reactive search control parameters.*/
  float mavg;              /* The moving average of the cycle length.*/
  int ssltlch;             /* The Steps Since the Last Tabu Length Change.*/
  unsigned long tourhv;  /*The hashing value of the current tour.*/
  long *z;   /* The vector of random numbers used to compute the hashing value of the tour.*/
  unsigned long h3t;          /* The intermediate hashing value of the tour.*/
  unsigned long  zin;         /* The values of the incoming tour hash function.*/
```

```
   unsigned long  zout;          /* The values of the outgoing tour hash function.*/
   struct hashlist *ptr;         /* A pointer to a tour if it is found.*/
   int esc_i, esc_d, esc_spi;    /* The "escape values" if all moves are tabu.*/
   long esc_best;                /* The objective value of the best of all moves.*/
   long atwl;                    /* average time window length */
   float HWB;
   double HWB1;
   double HWB2;

/*Initial input  values for search variables.*/
   k = 0;
   numfeas = 0;
   iter_no = 0;
   bfiter_no = 0;
   numdiff = 0;   bftour_cost = 999999;
   bftvl_time = 999999;
   feas_compl = 999999;
   num_veh_used = 0;
   num_feas_veh = 0;

/* Input the data from a data file */
/* Input the number of customers from the input file, including the depot.*/
   fscanf(ifp,"%d", &numcust);
   ++numcust; /* Increases numcust to incl the depot.*/
/*Determine the number of vehicles to be modeled.  If no vehicles are input, or number of
vehicles is more than the number of customers then model one vehicle for each customer.*/

   numveh = (numcust+1)/2;
   nnodes = numcust + numveh;  /* The total number of modeled nodes.*/

/* Lines to dynamically allocate memory for the problem vectors and matrices based on the
number of nodes actually in the problem.  */
   z = (long *)calloc(nnodes, sizeof(long));
   tour = (struct node *)calloc(nnodes, sizeof(NODE));
   initial_tour = (struct node *)calloc(nnodes, sizeof(NODE));
   best_tour = (struct node *)calloc(nnodes, sizeof(NODE));
   best_ftour = (struct node *)calloc(nnodes, sizeof(NODE));
   oldtour = (struct node *)calloc(nnodes, sizeof(NODE));
   oldtour2 = (struct node *)calloc(nnodes, sizeof(NODE));
   locx = (int *)calloc( nnodes, sizeof(int) );
   loc_initial = (int *)calloc( nnodes, sizeof(int) );
   time = (int **)calloc(nnodes, sizeof(int *));
   loctemp = (int *)calloc( nnodes, sizeof(int) );
   loctemp2 = (int *)calloc( nnodes, sizeof(int) );
   for (i=0; i < nnodes; ++i) time[i] = (int *)calloc(nnodes, sizeof(int));
   tabu_list = (int **)calloc(nnodes, sizeof(int *));
   for (i=0; i < nnodes; ++i) tabu_list[i] = (int *)calloc(nnodes-1, sizeof(int));
```

```
  input_pbm(numcust, numveh, g, tour, time, ifp, ofp);

/* Input the number of iterations, tabu length, and depth of search */
  mavg = nnodes-2;
  ssltlch = 0;
  tabu_length = min(30, nnodes-2);
  depth = nnodes-2;     /* Initialize the vector z[i] of random numbers.*/
  srand(1);
  for (i=0; i<numcust; ++i) {
    HWB1 = rand() ;
    HWB2 = rand()  ;
    HWB = (float) (HWB1*HWB2/32768.0/32768.0)  ;
    HWB = (float) (HWB * 131072.0 );
    z[i] = (long) (HWB+1.0) ;
  } /* The "new" way above avoids truncation problems
  z[i] = 1 + (long) (131072.0*(rand()/(RAND_MAX+1.0)));*/

/* Assign the same random hash number to all "dummy" vehicle nodes.*/
  for (i = numcust; i < nnodes; ++i) z[i] = z[0];

/* Initialize the hashing table to all NULL pointers.*/
  for (i = 0; i<HTSIZE; ++i) hashtbl[i] = NULL;

/*Conduct the time windows reduction for pairwise precedence.*/
  twredol(nnodes, tour, time, ofp);
  atwl = 0;
  for (i = 1; i<numcust; ++i) atwl += tour[i].l - tour[i].e;
  atwl = atwl/(numcust-1);
  printf("The average time window length (after time windows reduction is
    %ld\n", atwl);

/* DETERMINE INITIAL TOUR - Start Timing Here!*/
  start = clock();
/* This computes the initial schedule for the initial tour, and stores the values in the node
structure & returns the total tour length excluding any penalty for infeasibility.*/
  tour[0].arr = tour[0].e;
  tour[0].dep = tour[0].e; /* Initialize starting values at the depot.*/
  tour[0].wait = 0;    /* These will never change for this model.*/
  tour[0].load = 0;
  build_tour(tour, time, numcust, nnodes, CAP);
  num_veh_used = count_vehs(nnodes, tour);
  nnodes = numcust+num_veh_used+1;
 /* Determine and store where the nodes are located in the tour */
  for ( i = 0; i < nnodes; ++i ) locx[tour[i].id] = i;

 /* Determine the customer vehicle assignment */
  for (i=nnodes-1; i > 0; --i) {
```

```
    if (tour[i].type == 2) {
      vehicle_id = tour[i].id;
      tour[i].vehicle = vehicle_id;
    }
    if (tour[i].type == 1) tour[i].vehicle = vehicle_id;
  }
/* Determine the time required to compute the initial tour */
  init_end = clock();
  init_time = (((double)init_end - (double)start)/(double)CLOCKS_PER_SEC);
/* Compute the length of the tour */
  tour_length = tour_sched(1, nnodes, tour, time);
/* Calculate the initial infeasibilities.*/
  comp_parpens(&tourpen, nnodes, tour, CAP);
  tot_penalty  = tourpen.tw + tourpen.ld ;
  if (tot_penalty > 0) {
    fprintf(ofp,"This starting tour is infeasible!\n");
    bt_pen_cost = tot_penalty;
  } /*Initialize the parameter.*/
  else bt_pen_cost = 999999;
/* Compute the infeasibility costs.*/
  tw_cost =  TWPEN*tourpen.tw ;
  load_cost =  VPEN*tourpen.ld ;
  pen_cost = tw_cost + load_cost ;

/* Compute the intial tour values.*/
  tour_cost = tour_length + pen_cost;
  totwait = sum_wait(nnodes, tour);
  pen_tt = tour_cost - totwait;
  tvl = pen_tt - pen_cost;
/* Retain the initial values as the best tour found values.*/
  best_cost = pen_tt;
  best_tt = tvl;
  best_time = 0.0;
  bestf_time = 999999.0;
  for (i= 0; i< nnodes; ++i) best_tour[i] = tour[i];
  fprintf(ofp,"The tour cost is %9.1f, the travel time is %9.1f.\n",
    (float)(tour_cost/FACTOR), (float) (tvl/FACTOR));
/* Open, Print the initial tour data to file "initial.out", close the file.*/
  fpinit = fopen("wbinit.out", "a");
/* Count the vehicles used, see file "makespan.h"*/
  num_veh_used = count_vehs(nnodes, tour);
  fprintf(fpinit," %6.1f   %3d   %6.2f\n", (double) tvl/FACTOR,num_veh_used, init_time);
  fclose(fpinit);
 /* Compute the hashing value for the initial tour.*/
  h3t = 0;
  for (i = 0; i < nnodes-1; ++i) h3t += z[tour[i].id]*z[tour[i+1].id];
  tourhv = h3t;
```

```
/* The tabu search subroutine evaluates all insert neighbors and finds the best to change and
outputs a new tour and schedule at every iterations. */
   for (i = 0; i< nnodes-1; ++i)
   for (j = 0; j< nnodes-1; ++j) tabu_list[i][j] = 0;

/* Initialize tabu structure. */
   fprintf(ofp,"\nTabu_length is %d and the Number of iterations is %d. \n", tabu_length, niters);
   fprintf(ofp,"The depth of the search is %d.\n", depth);
   ++k;  /* Increment k */
   if (tot_penalty == 0) keep_bfs(nnodes, tour, tour_cost, tvl, best_ftour, &bftour_cost,
     &bftvl_time, &bfiter_no, k-1, start, &bestf_time);
   while (k <= niters) {
     printf("\r%d",k); fflush(stdout);

/* Look for the incumbent tour in the hashing structure.  This is the effort to determine if any
tours are repeated and to adjust the tabu_length accordingly.*/
   ptr = lookfor(tour_cost, tourhv, tourpen.tw, tourpen.ld, k-1, tvl);
   if (tot_penalty == 0 &&  ptr == NULL) ++numfeas;
   if (ptr == NULL) {
     notfound(&tabu_length, &ssltlch, mavg, ofp);
     ++numdiff;
   }
   else {
     found(ptr, &tabu_length, &ssltlch, &mavg, k, ofp);
     if (ptr->repeat == 2) ++revisit;
   }
/* Reset the search parameters to initiate a new search.*/
   d_best = esc_best = d_bestf = 999999;
   ch_i = feas_i = esc_i = 0;
   ch_d = feas_d = esc_d = 0;
   ch_spi = feas_spi = esc_spi = 0;
   same = 0;

/* Perform the multineighborhood strategic search (hierarchical) methodology */
   if (k == niters/2 && restart == 0) {
   for (i = 0; i <= nnodes-1; ++i) {
     tour[i] = best_tour[i];
     locx[tour[i].id] = i;
   }
   num_veh_used = count_vehs(nnodes, tour);
   mavg = nnodes-2;
   ssltlch = 0;
   tabu_length = min(30, nnodes-2);
   depth = nnodes-2;
   for (i = 0; i<HTSIZE; ++i) hashtbl[i] = NULL;
   for (i = 0; i< nnodes-1; ++i)
     for (j = 0; j< nnodes-1; ++j) tabu_list[i][j] = 0;
```

```
    printf("\nForcing a restart at best tour.\n");
    fprintf(ofp,"\nForcing a restart at best tour.\n");
    revisit = 0;
    current = 0;
    q = 0;
  }
    if (revisit >= 10 && q == 0) { /* caught in an attractor basin */
      ++next;
      if (next <= min(num_veh_used,numcust/10)) goto swap_pairs;
      else if (next > min(num_veh_used,numcust/10)) {
        printf("\nLocked in a chaotic attractor basin.\n");
        fprintf(ofp,"\nLocked in a chaotic attractor basin.\n");
        next = 0;
        revisit = 0;
        current = 1;
        q = 1;
        restart = 1;
      }
    }
    if (revisit >= 10 && q == 1) {
      for (i = 0; i <= nnodes-1; ++i) {
        tour[i] = best_tour[i];
        locx[tour[i].id] = i;
      }
      num_veh_used = count_vehs(nnodes, tour);
      mavg = nnodes-2;
      ssltlch = 0;
      tabu_length = min(30, nnodes-2);

      depth = nnodes-2;
      for (i = 0; i<HTSIZE; ++i) hashtbl[i] = NULL;
      for (i = 0; i< nnodes-1; ++i)
        for (j = 0; j< nnodes-1; ++j) tabu_list[i][j] = 0;
      revisit = 0;
      current = 0;
      q = 0;
    }

    if (tour[0].l/atwl <= 4) { /* many feasible solutions */
      if (current == 0) {
        ++current;
        goto SPI;
      }
      if (current <= numcust/10) {
        ++current;
        goto WRI;
      }
```

```
    if (current > numcust/10) {
      current = 1;
      goto SPI;
    }
  }
  if (tour[0].l/atwl > 4) {
    if (current == 0) {
      ++current;
      goto SPI;
    }
    if (current <= numcust/25) {
      ++current;
      goto WRI;
    }
    if (current > numcust/25) {
      current = 1;
      goto SPI;
    }
  }

/* The following candidate list looks for improvement within the respective routes only. */

/* Conduct LATER Within Route Insertions (WRI) */
WRI:
++wricount;
for (i=1; i< nnodes-2; ++i) {
  if (tour[i].type == 2) continue; /* You are at a vehicle node */
  if ((tour[i].id == locx[i] == i) && tour[i].type == 2) break;
  for (j=0; j < nnodes ; ++j)  {
    oldtour[j] = tour[j];
    loctemp[j] = locx[j];
  }
  for (d = 1; d <= depth; ++d) {
    if (i+d == nnodes - 1) break; /* end of tour */
    if (tour[i].vehicle != tour[i+d].vehicle) break;
/* Not on the same route */
    else { /* search along same route */
      if (tour[i+d].pred == tour[i].id || tour[i].succ == tour[i+d].id || tour[i+d].e +
        time[tour[i+d].id][tour[i].id]> tour[i].l) break;
/* Do not change a precedent viable pair or if the swap strongly violates time windows */
    else {
      SWAP(loctemp[oldtour[i+d-1].id], loctemp[oldtour[i+d].id], temp1);
      swap_node(i+d-1, i+d, oldtour);
      move_val = move_delta(i, d, nnodes, tour, oldtour, time);
      comp_parpens(&nbrpen, nnodes, oldtour, CAP);
      tot_nbrpen = nbrpen.tw + nbrpen.ld ;
      move_val += TWPEN*(nbrpen.tw - tourpen.tw) + VPEN*(nbrpen.ld - tourpen.ld);
```

```
      } /* end else */
      if (tot_nbrpen == 0) {
/* If tot_nbrpen = 0 => feasible neighbor tour.*/
        if (move_val < d_bestf) {
          if (k > tabu_list[tour[i].id][i+d] || (move_val+pen_tt < best_cost)) {
            d_bestf = move_val;
            feas_i = i;
            feas_d = d;
          } /* end if not tabu */
        } /* end if improved move value (travel time) */

      } /* end if feasible neighbor found.*/
      else {
      if (move_val < d_best) {
        if (k > tabu_list[tour[i].id][i+d] || (move_val+pen_tt < best_cost)) {
          d_best = move_val;
          ch_i = i;
          ch_d = d;
        } /* end if not tabu */
        } /* end if improved move value */
      } /* end else: infeasible neighbor */ /* escape routine */
      if (move_val < esc_best) {
/* Finds the best of all neighboring moves regardless if tabu or not.*/
        esc_best = move_val;
        esc_i = i;
        esc_d = d;
      } /* end escape if */
      } /* end else customers on same route*/
    } /* end for d */
  } /* end for i */

  /* Check all EARLIER Within Route Insertions */

  for (i=3; i<=nnodes-2; ++i) {
    if (tour[i].type == 2) continue; /* You are at a vehicle node */
    if ((tour[i].id == locx[i] == i) && tour[i].type == 2) break;
    for (j=0; j < nnodes ; ++j) {
      oldtour[j] = tour[j];
      loctemp[j] = locx[j];
    }
    for (d = 1; i > d; ++d) {
      if (tour[i-d].type == 2) break;
/* not on same vehicle or you are at the depot */
      else {
        if (tour[i].pred == tour[i-d].id || tour[i-d].succ == tour[i].id || tour[i].e +
          time[tour[i].id][tour[i-d].id] > tour[i-d].l) break;
      else {
```

```
        SWAP(loctemp[oldtour[i-d].id], loctemp[oldtour[i-d+1].id], temp1);
        swap_node(i-d, i-d+1, oldtour);
        move_val = move_delta(i, -d, nnodes, tour, oldtour, time);
        comp_parpens(&nbrpen, nnodes, oldtour, CAP);
        tot_nbrpen = nbrpen.tw + nbrpen.ld ;
        move_val += TWPEN*(nbrpen.tw - tourpen.tw) + VPEN*(nbrpen.ld - tourpen.ld);
      } /* end else */
      if (tot_nbrpen == 0) {
/*If tot_nbrpen == 0 => feasible neighbor tour.*/
        if (move_val < d_bestf)  {
          if (k > tabu_list[tour[i].id][i-d] || (move_val+pen_tt < best_cost)) {
            d_bestf = move_val;
            feas_i = i;
            feas_d = -d;
          } /* end if not tabu*/
        }/*end if improved move value (travel time).*/

      } /*end if feasible neighbor found.*/
      else {
      if (move_val < d_best)  {
        if (k > tabu_list[tour[i].id][i-d] || (move_val+pen_tt<best_cost)) {
          d_best = move_val;
          ch_i = i;
          ch_d = -d;
        } /* end if tabu*/
      } /* end if improved move value*/
      }/* end else: infeasible neighbor.*/ /*escape routine*/
      if (move_val < esc_best)  {  /* Finds the best neighboring move.*/
        esc_best = move_val;
        esc_i = i;
        esc_d = -d;
      }  /* end escape if */
    } /* end else: customer node */
    } /* end for d*/
  } /*end for i*/

  /* If there are no moves to be made, move to the next search scheme */

  if (esc_i == 0) {
    --wricount;
    goto SPI;
  }

  /* If a feasible move is found...move to it!*/
  if (feas_i != 0) {
    ch_i = feas_i;
    ch_d = feas_d;
```

```
} /* end if feasible move is found.*/

/* If all moves are tabu and none meet the aspiration criteria, then set ch_i and ch_d to the best
move discovered and decrease the tabu length. */

if(ch_i == 0 && esc_i != 0)  {
   ch_i = esc_i;
   ch_d = esc_d;
   tabu_length = max((int) (tabu_length)*DECREASE,5);
}/* end for all moves tabu*/

/* UPDATE: TABU LIST AND TOUR POSITIONS    Allows no "return" moves for tabu_length
iterations. */
   if (ch_d == 1) tabu_list [tour[ch_i+1].id][ch_i+1] = k + tabu_length;
   else tabu_list [tour[ch_i].id][ch_i] = k + tabu_length;
/* Allows no "repeat" moves for tabu_length iterations. */
    tabu_list [tour[ch_i].id][ch_i+ch_d] = k + tabu_length;
/* BEFORE the new tour is constructed, update the h3t value:*/
   zin = zout = 0;
   i= ch_i;
   j=((ch_d>0) ? ch_i+ch_d : ch_i+ch_d-1);
   zout = z[tour[i-1].id]*z[tour[i].id] + z[tour[i].id]*z[tour[i+1].id] + z[tour[j].id]*z[tour[j+1].id];
   zin  = z[tour[i-1].id]*z[tour[i+1].id] + z[tour[j].id]*z[tour[i].id]+ z[tour[i].id]*z[tour[j+1].id];
   h3t += zin - zout;
   tourhv = h3t;
   tour = insert(ch_i, tour, locx, ch_d, nnodes);
   tour_length = ((ch_d >0) ? tour_sched(ch_i, nnodes, tour, time)
     : tour_sched(ch_i+ch_d, nnodes, tour, time));
   goto update;

SPI:
/* Now for the more rigorous SPI. This is an O(n^2) search for single pair insertions. We initially
select predecessors to move. We know where there successors are located. We only look to
move the pair on other routes. This will help expedite the search. */
++spicount;
for (i=1; i< nnodes-2; ++i) {
   if ((tour[i].vehicle == i) && tour[i].type == 2) break;
/* search starts only with a predecessor node */
   if (tour[i].type == 2 || tour[i].pred != 0) continue;
   for (j=1; j<nnodes-2; ++j) {
     if (tour[i].vehicle != tour[j].vehicle) { /* place on another route */
       if (tour[j].type == 2) {
/* if you are attempting to insert before the last node on the route, the vehicle node, you must
check to see if you can insert the pairwise-POS. Checking to insert the predecessor (supply node)
before the vehicle (depot) node is not permitted. It is one of the inadmissable arcs. This checks
SPATIAL fit */
```

```
    if (tour[j-1].dep + time[tour[j-1].id][tour[i].id] + time[tour[i].id][tour[i].succ]
      + time[tour[i].succ][tour[j].id] <= tour[j].l && tour[i].e +
      time[tour[i].id][tour[i].succ] + time[tour[i].succ][tour[j].id] <= tour[j].l)
      goto updatetour;
    } /* end if insertion before vehicle node */
/* insert BEFORE the j position if it does not violate strong TWs. The dominance of precedence
is established by where we can possibly insert the successor */
    if (tour[j-1].dep + time[tour[j-1].id][tour[i].id] + time[tour[i].id][tour[j].id]
      <= tour[j].l && tour[i].e + time[tour[i].id][tour[j].id] <= tour[j].l) {
updatetour:
      current_vehicle = tour[j].vehicle;
      for (j2=0; j2 < nnodes ; ++j2)  {
        oldtour[j2] = tour[j2];
        loctemp[j2] = locx[j2];
      } /* end for j2 - setting up working tour */
/* Insert the predecessor */
      oldtour = ((i<j) ? insert(i, oldtour, loctemp,j-i-1,nnodes)
        : insert(i, oldtour, loctemp, j-i, nnodes));
/* Compute an initial change in the tour - this change IGNORES precedence and coupling */
      move_val = ((i<j) ? move_delta(i, j-i-1, nnodes, tour, oldtour, time)
        :  move_delta(i, j-i, nnodes, tour, oldtour, time));
      for (d=0; d<=depth && tour[j+d].vehicle == current_vehicle; ++d) {
/* Start with d=0 so you can insert the successor immediately following the predecessor */
        if (tour[i+d].vehicle == i+d && tour[i+d].type == 2) break;
/* Need to save the passing of the predecessor for oldtour in preparation for the move of the
successor */
        for (d2=0; d2 < nnodes; ++d2) {
          oldtour2[d2] = oldtour[d2];
          loctemp2[d2] = loctemp[d2];
        } /* end for d2 */

/* By evaluating ALL possible insertions after the predecessor, we can force exploration of
infeasible solutions. Some of these may have better travel times!*/
        oldtour2 = ((i<j) ? insert(loctemp[tour[i].succ], oldtour2, loctemp2,
          j+d-loctemp[tour[i].succ]-1,nnodes) : insert(loctemp[tour[i].succ],
          oldtour2, loctemp2, j+1+d-loctemp[tour[i].succ], nnodes));
        move_val2 = ((i<j) ? move_delta(loctemp[tour[i].succ],
          j+d-loctemp[tour[i].succ]-1, nnodes, oldtour, oldtour2, time)+move_val
            : move_delta(loctemp[tour[i].succ], j+1+d-loctemp[tour[i].succ],
            nnodes, oldtour, oldtour2, time) + move_val);
        comp_parpens(&nbrpen, nnodes, oldtour2, CAP);
        tot_nbrpen = nbrpen.tw + nbrpen.ld ;
/* Note: precedence and coupling constraints are satisfied. */
        move_val2 += TWPEN*(nbrpen.tw - tourpen.tw) + VPEN*(nbrpen.ld - tourpen.ld);
  if (i<j) { /* insert LATER */
    if (tot_nbrpen == 0) {
/* If tot_nbrpen = 0 => feasible tour.*/
```

```
        if (move_val2 < d_bestf) {
        if (k > tabu_list[tour[i].id][j-1] ||
          (move_val2+pen_tt < best_cost)) {
          d_bestf = move_val2;
          feas_i = i; /* which predecessor is moving */
          feas_spi = j; /* where pred moves before */
          feas_d = d;  /* how much further away the successor is placed */
        } /* end if not tabu */
        } /* end if improved move value (travel time) */

      } /* end if feasible neighbor found.*/
      else {
        if (move_val2 < d_best) {
        if (k > tabu_list[tour[i].id][j-1] || (move_val2+pen_tt < best_cost)) {
          d_best = move_val2;
          ch_i = i;
          ch_spi = j;
          ch_d = d;
        } /* end if not tabu */
        } /* end if improved move value */
      } /* end else: infeasible neighbor */
    } /* end if i<j */
    else { /* insert EARLIER */
      if (tot_nbrpen == 0) {
/* If tot_nbrpen = 0 => feasible neighbor tour.*/
        if (move_val2 < d_bestf) {
          if (k > tabu_list[tour[i].id][j] || (move_val2+pen_tt < best_cost)) {
            d_bestf = move_val2;
            feas_i = i;
            feas_spi = j;
            feas_d = d;
          } /* end if not tabu */
      } /* end if improved move value (travel time) */

    } /* end if feasible neighbor found.*/
    else {
      if (move_val2 < d_best) {
        if (k > tabu_list[tour[i].id][j] || (move_val2+pen_tt < best_cost)) {
          d_best = move_val2;
          ch_i = i;
          ch_spi = j;
          ch_d = d;
        } /* end if not tabu */
      } /* end if improved move value */
    } /* end else: infeasible neighbor */
          } /* end else */
        } /* end for d */
```

```
        } /* end if predeccessor fits */
      } /* end if on different routes */
    } /* end for j */
  } /* end for i - finished rigorous search */

  /* If a feasible move is found...move to it!*/
  if (feas_i != 0) {
    ch_i = feas_i;
    ch_spi = feas_spi;
    ch_d = feas_d;
  } /* end if feasible move is found.*/

  /* Note: if ch_i == 0, there is no improving pair to insert */
  if (ch_i == 0) {
    --spicount;
    d_best = esc_best = d_bestf = 999999;
    ch_i = feas_i = esc_i = 0;
    ch_d = feas_d = esc_d = 0;
    ch_spi = feas_spi = esc_spi = 0;
    goto swap_pairs;  /* escape routine */
  }
  if (ch_i != 0) {
    same = 1;  /* found location to insert pair */
    esc_i = ((ch_i < ch_spi) ? locx[tour[ch_i].succ] - 1 : locx[tour[ch_i].succ]);
    esc_d = ((ch_i < ch_spi) ? ch_spi+ch_d-esc_i-1 : ch_spi+1+ch_d-esc_i);

  /* Need to check that if we are reducing the number of vehicles, that we only perform the
  reduction if the move is to a feasible tour.  We will not consider an infeasible move until most of
  the search has been completed. */
    for (j2=0; j2 < nnodes ; ++j2)  {
      oldtour[j2] = tour[j2];
      loctemp[j2] = locx[j2];
    } /* end for j2 - setting up working tour */
    oldtour = ((ch_i<ch_spi) ? insert(ch_i, oldtour, loctemp, ch_spi-ch_i-1, nnodes)
      : insert(ch_i, oldtour, loctemp, ch_spi-ch_i, nnodes));
    if (same) { /* insertion of a pair satisfying (a) or (b) on another route */
  /* insert the successor from second neighborhood search */
      if (ch_i<ch_spi) oldtour = insert(esc_i, oldtour, loctemp, esc_d, nnodes);
      else oldtour = insert(esc_i, oldtour, loctemp, esc_d, nnodes);
      tour_length = tour_sched(1, nnodes, oldtour, time);
    }
    comp_parpens(&nbrpen, nnodes, oldtour, CAP);
    tot_penalty = nbrpen.tw + nbrpen.ld ;
    if (count_vehs(nnodes, oldtour) < num_veh_used) {
    if (tot_penalty == 0 || k > .5*niters) num_veh_used = count_vehs(nnodes, oldtour);
    if (tot_penalty != 0 && k <= .5*niters) {
      --spicount;
```

```
      d_best = esc_best = d_bestf = 999999;
      ch_i = feas_i = esc_i = 0;
      ch_d = feas_d = esc_d = 0;
      ch_spi = feas_spi = esc_spi = 0;
      same = 0;
      goto swap_pairs;
    }
    }
/* UPDATE: TABU LIST AND TOUR POSITIONS */
/* Allows no "return" moves for tabu_length iterations. Uses the predecessor only for the second
neighborhood search scheme */
    tabu_list [tour[ch_i].id][ch_i] = k + tabu_length;
/* Allows no "repeat" moves for tabu_length iterations. */
    if (ch_i<ch_spi) tabu_list [tour[ch_i].id][ch_spi-1] = k + tabu_length;
    else tabu_list [tour[ch_i].id][ch_spi] = k + tabu_length;
/* BEFORE the new tour is constructed, update the h3t value:*/
    zin = zout = 0;
    i= ch_i;
    j=((ch_i < ch_spi) ? ch_spi-1 : ch_spi);
    zout = z[tour[i-1].id]*z[tour[i].id] + z[tour[i].id]*z[tour[i+1].id] + z[tour[j].id]*z[tour[j+1].id];
    zin  = z[tour[i-1].id]*z[tour[i+1].id] + z[tour[j].id]*z[tour[i].id]+ z[tour[i].id]*z[tour[j+1].id];
    h3t += zin - zout;
    tourhv = h3t;
    for (j2=0; j2 < nnodes ; ++j2)  {
      tour[j2] = oldtour[j2];
      locx[j2] = loctemp[j2];
    } /* end for j2 - setting up working tour */

    tour_length = tour_sched(1, nnodes, tour, time);
    first = 1;
    goto update;
} /* end if found insert pair */

swap_pairs:
++swapcount;
/* move to the third search neighborhood where we swap pairs between routes */
if (ch_i == 0) {
    for (i = 1; i < nnodes-2; ++i) {
    if (tour[i].type == 2) continue; /* You are at a vehicle node */
    if ((tour[i].id == locx[i] == i) && tour[i].type == 2) break;
    if (tour[i].succ != 0) {/* if it has a successor, then this node is a predecessor */
    for (j = i+1; j < nnodes-2; ++j) {
    if (tour[j].vehicle != tour[i].vehicle && tour[j].succ != 0) {
      for (d=0; d < nnodes ; ++d)  {
        oldtour[d] = tour[d];
        loctemp[d] = locx[d];
      } /* end for d - setting up working tour */
```

```
      if (k > tabu_list[tour[i].id][j] && k > tabu_list[tour[j].id][i]) {
        SWAP(oldtour[loctemp[oldtour[i].succ]].vehicle,
          oldtour[loctemp[oldtour[j].succ]].vehicle, temp1);
        SWAP(oldtour[loctemp[oldtour[i].succ]], oldtour[loctemp[oldtour[j].succ]], temp2);
        SWAP(loctemp[oldtour[i].succ], loctemp[oldtour[j].succ],temp1);
        SWAP(oldtour[i].vehicle, oldtour[j].vehicle, temp1);
        SWAP(oldtour[i], oldtour[j], temp2);
        SWAP(loctemp[oldtour[i].id], loctemp[oldtour[j].id], temp1);
        tour_length = tour_sched(i, nnodes, oldtour, time);
        comp_parpens(&nbrpen, nnodes, oldtour, CAP);
        tot_nbrpen = nbrpen.tw + nbrpen.ld ;
        move_val = tour_length + TWPEN*nbrpen.tw + VPEN*nbrpen.ld;
        if (tot_nbrpen == 0) {
/* If tot_nbrpen = 0 => feasible tour.*/
          if (move_val < d_bestf) {
            d_bestf = move_val;
/* which predecessor is moving */
            feas_i = i;
/* where predecessor moves before */
            feas_spi = j;
          } /* end if improved move value (travel time) */
        } /* end if feasible neighbor found.*/
        else {
          if (move_val < d_best) {
            d_best = move_val;
            ch_i = i;
            ch_spi = j;
          } /* end if improved move value */
        } /* end else: infeasible neighbor */ /* escape routine */
        if (move_val < esc_best) {
/* Finds the best of all neighboring moves regardless if tabu or not.*/
          esc_best = move_val;
          esc_i = i;
          esc_spi = j;
        }  /* end escape if */
      } /* end tabu check */
    } /* end if on different vehicles - predecessors identified */
    } /* end for j */
} /* end if predecessor identified */ } /* end for i */
if (feas_i != 0) { /* feasible neighbor - move to it */
    ch_i = feas_i;
    ch_spi = feas_spi;
}
if (ch_i == 0) {
    ch_i = esc_i;
    ch_spi = esc_spi;
}
```

```
if (ch_i != 0) { /* perform the swap... */
   SWAP(tour[locx[tour[ch_i].succ]].vehicle, tour[locx[tour[ch_spi].succ]].vehicle, temp1);
   SWAP(tour[locx[tour[ch_i].succ]], tour[locx[tour[ch_spi].succ]], temp2);
   SWAP(locx[tour[ch_i].succ], locx[tour[ch_spi].succ], temp1);
   SWAP(tour[ch_i].vehicle, tour[ch_spi].vehicle, temp1);
   SWAP(tour[ch_i], tour[ch_spi], temp2);
   SWAP(locx[tour[ch_i].id], locx[tour[ch_spi].id], temp1);
/* ... and update the tabu structure */
   tabu_list [tour[ch_i].id][ch_i] = k + tabu_length;
   tabu_list [tour[ch_spi].id][ch_spi] = k + tabu_length;
/* prevents direct (active) move back into the position into which the node is current moving */
   tabu_list [tour[ch_i].id][ch_spi] = k + tabu_length;
   tabu_list [tour[ch_spi].id][ch_i] = k + tabu_length;

/* BEFORE the new tour is constructed, update the h3t value: this is based on swapping the
predecessor at position with the predecessor at position j */
   zin = zout = 0;
   i = ch_i;
   j = ch_spi;
   zout = z[tour[i-1].id]*z[tour[i].id] + z[tour[i].id]*z[tour[i+1].id]+ z[tour[j].id]*z[tour[j+1].id];
   zin  = z[tour[i-1].id]*z[tour[i+1].id] + z[tour[j].id]*z[tour[i].id]+ z[tour[i].id]*z[tour[j+1].id];
   h3t += zin - zout;
   tourhv = h3t;
   tour_length = tour_sched(1, nnodes, tour, time);
   goto update;  /* update the tour */
} /* end if */ } /* end if ch_i == 0 */
/* Calculate the new incumbent tour infeasibilities.*/
update:
   comp_parpens(&tourpen, nnodes, tour, CAP);
/* Compute the infeasibility costs */
   tot_penalty = tourpen.tw + tourpen.ld ;
   tw_cost = TWPEN*tourpen.tw;
   load_cost = VPEN*tourpen.ld;
   pen_cost = tw_cost + load_cost ;
/* Compute the new incumbent tour values.*/
   tour_cost = tour_length + pen_cost;
   totwait = sum_wait(nnodes, tour);
   pen_tt = tour_cost - totwait;
   tvl = pen_tt - pen_cost;

   if (tot_penalty == 0)
      keep_bfs(nnodes, tour, tour_cost, tvl, best_ftour, &bftour_cost,
         &bftvl_time,  &bfiter_no, k, start, &bestf_time);
   if (pen_tt < best_cost) {
      best_tt = tvl;
      bt_pen_cost = tourpen.tw + tourpen.ld;
      best_cost = pen_tt;
```

```
      soln_time = clock();
      best_time = (((double)soln_time - (double)start)/(double)CLOCKS_PER_SEC);
      for (i=1; i< nnodes; ++i) best_tour[i] = tour[i];
      iter_no = k;
      if (k > niters/4) {
      fprintf(ofp,"\n\nA (possible) super-optimal solution was found on the %dth iteration.\n",
        iter_no);
      print_sched(best_tour, nnodes, ofp);
      fprintf(ofp,"\nThe travel time of this tour is%8.1f.\n", (float) best_tt/FACTOR);
      fprintf(ofp,"The time windows violation = %ld\n", tourpen.tw);
      ++count;
    }
  } /* end if -- end the update of the best tour value & best tour.*/
    ++k;
  } /* end while....Ends the tabu search subroutine */

  /*Add the tour found at the last iteration to the hash table, if necessary.*/
    ptr = lookfor(tour_cost, tourhv, tourpen.tw, tourpen.ld, k-1, tvl);
    if (tot_penalty == 0 && ptr == NULL) ++numfeas;
    if (ptr == NULL) notfound(&tabu_length, &ssltlch, mavg, ofp);
    else found(ptr, &tabu_length, &ssltlch, &mavg, k, ofp);
    stop = clock();
    duration = (((double)stop - (double)start)/(double) CLOCKS_PER_SEC);
    if (bestf_time != 999999.0) makespan(nnodes, best_ftour, &feas_compl, &num_feas_veh);
    makespan(nnodes, best_tour, &compl_time, &num_veh_used);

  /* Output the best tour found, iteration number, tour length, the shortest tour found overall
  regardless of feasibility and the number of feasible tours discovered during the search. */

  /* Record the solution values for the summary sheet.*/
    soln[0] = (float)(((bestf_time != 999999.0)? feas_compl: compl_time)/FACTOR);
    soln[1] =(float) (((bestf_time != 999999.0)? bftvl_time: best_tt)/FACTOR);
    soln[2] = (float) (bfiter_no);
    soln[3] = (float)(best_tt/FACTOR);
    soln[4] = (float) (((bftvl_time == best_tt)? 0: bt_pen_cost)/FACTOR);
    soln[5] =  (float) (iter_no);
    soln[6] = (float) (((bestf_time != 999999.0)? bestf_time: best_time));
    soln[7] = (float) (numfeas);
    soln[8] = (float) (compl_time/FACTOR);
    soln[9] = (float) (duration);
    soln[10] = (float) (((bestf_time != 999999.0)? bfiter_no: iter_no));
    soln[11] = (float) (feas_compl/FACTOR);
    soln[12] = (float) (num_feas_veh);
    soln[13] = (float) (num_veh_used);
    soln[14] = (float) (0);
    if (bftour_cost < 999999)  {
      fprintf(ofp,"The best feasible tour was found on the %dth iteration.\n", bfiter_no);
```

```
    print_sched(best_ftour, nnodes, ofp);
    fprintf(ofp,"The cost of the tour is %ld.\n", (long) (bftour_cost/FACTOR));
    fprintf(ofp,"The travel time of the tour is %ld.\n", (long) (bftvl_time/FACTOR));
  }
  else fprintf(ofp,"THE ALGORITHM FOUND NO FEASIBLE TOUR.");
  if (best_tt != 0 /*bftvl_time*/) {
    fprintf(ofp,"\n\nThe best overall travel time tour was found on the %dth iteration.\n",
      iter_no);
    print_sched(best_tour, nnodes, ofp);
    fprintf(ofp,"The length of the tour is %8.1f.\n", (float) compl_time/FACTOR);
    fprintf(ofp,"\nThe travel time of this tour is%8.1f.\n\n", (float) best_tt/FACTOR);
  }
  else fprintf(ofp,"THE BEST TOUR FOUND IS THE FEASIBLE TOUR ABOVE!\n\n");
  fprintf(ofp,"The tabu search routine took %.3f seconds.\n\n", duration);
  fprintf(ofp,"The search found a total of %d feasible tours.\n", numfeas);
  fprintf(ofp,"The search found a total of %d different tours.\n", numdiff);
  fprintf(ofp,"wricount=%d\tspicount=%d\tswapcount=%d\n",wricount, spicount, swapcount);
  fprintf(ofp,"The average time window length = %ld\n", atwl);
  fprintf(ofp,"The planning horizon = %d\n", tour[0].l); fprintf(ofp,"\nThere
    were %d (possible) super-optimal tours\n", count);
  printf("\nThere were %d (possible) super-optimal tours\n", count);

/* Free all the memory structures after every problem to start over every time */
  free(z);
  free(tour);
  free(initial_tour);
  free(best_tour);
  free(best_ftour);
  free(oldtour);
  free(oldtour2);
  free(locx);
  free(loc_initial);
  free(loctemp);
  free(loctemp2);
  for (i=0; i < nnodes; ++i) free(time[i]);
  free(time);
  for (i=0; i < nnodes; ++i) free(tabu_list[i]);
  free(tabu_list);

  return &soln[0];

}/* end of the t_search function */

/* This function computes the incremental change in the value of the incumbent tour to the
proposed neighbor tour.*/
int move_delta(i, d, n, t, tt, ti)
  int i;          /* The starting point for computing the value.*/
```

```
  int d;          /* The depth of the insertion.*/
  int n;          /* The number of nodes in the tour.*/
  NODE *t;        /* The incumbent tour structure.*/
  NODE *tt;       /* The neighbor (temporary) tour structure.*/
  int **ti;       /* The time/distance matrix.*/
{
  int is;         /* The starting point for the nbr schedule.*/
  int j;          /* The index of the "target" node of insertion.*/
  int delin;      /* The incremental tour travel time.*/
  int delout;     /* The incremental tour travel time.*/
  NODE *npi, *npi1;  /* Node pointer indexes used to iterate.*/

/* This routine stops computing when a vehicle node is encountered after the "within" area of
change.*/
  int iend;         /* Index to end of "within" area of insertion.*/
  NODE *npe;        /* Pointer to the end of within insertion.*/
  delin = delout = 0; if (d>0) {
    j = i+d;
    is = i+d-1;
    iend = i+d+1;
  }/* end if */
  else {
    j = i+d-1;
    is = i+d;
    iend = i+d+3;
  } /* end else */
  npi = &tt[is-1];
  npe = &tt[iend];
/* This is a procedure for updating the schedule from istart to the appropriate vehicle node, or the
terminal depot.*/
  while (npi < npe || npi->type == 1) {
    npi1 = npi+1;
    npi1->load = ((npi->type == 2)? 0: npi->load) + npi1->qty;
    npi1->arr = npi->dep + ti[npi->id][npi1->id];
    if (npi1->type == 2) {
      npi1->dep = npi1->e;
      npi1->wait = 0;
    } /* end if vehicle node */
    else {
      npi1->dep = max(npi1->e, npi1->arr);
      npi1->wait = npi1->dep - npi1->arr;
    } /* end if customer node */
    ++npi;
  } /* end while */
  delout = ti[t[i-1].id][t[i].id] + ti[t[i].id][t[i+1].id] + ti[t[j].id][t[j+1].id];
  delin = ti[t[i-1].id][t[i+1].id] + ti[t[j].id][t[i].id] + ti[t[i].id][t[j+1].id];
```

```
  return (delin - delout);
} /* This ends the computation of the move value.*/
long length;
/* This function computes the tour schedule for a neighbor (temporary) tour.  It returns the total
value of the tour length. */
long tour_sched(istart, n, t, ti)
  int istart;        /*The starting point for computing the sched.*/
  int n;             /*The number of nodes in the tour.*/
  NODE *t;           /*The tour structure.*/
  int  **ti;         /*The time-distance matrix.*/
{
  NODE *h;              /*Index for the pointer to istart-1.*/
  NODE *lastnp;         /*Index for the pointer to the last node.*/
  NODE *np, *np1;       /*Node pointer indexes used to iterate.*/
  long tour_length;     /*The total tour length.*/
  long length;
  tour_length = 0;
  h = &t[istart-1];
  lastnp = &t[n-1];

/*This computes the tour length from the origin depot to istart.*/
  for (np = &t[0]; np < h; ++np) tour_length +=  ti[np->id][(np+1)->id] + (np+1)->wait;

/* This is a procedure for updating the schedule from istart to the terminal depot.*/
  for (np = h; np < lastnp; ++np) {
    np1 = np+1;
    np1->load = ((np->type == 2)? 0: np->load) + np1->qty;
    np1->arr = np->dep +ti[np->id][np1->id];
    if (np1->type == 2) {
      np1->dep = np1->e;
      np1->wait = 0;
    } /* end if vehicle node */
    else {
      np1->dep = max(np1->e, np1->arr);
      np1->wait = np1->dep - np1->arr;
    } /* end else customer node.*/
    tour_length +=  ti[np->id][np1->id] + np1->wait;
  } /* end for */
  length = tour_length;

  return (length);
} /* This ends the computation of the tour schedule.*/

void comp_parpens(penptr, n, t, c)
/* This is used to compute the time window and load penalties. */
  PENALTY *penptr;    /* The pointer to the penalty structure.*/
  int n;              /* The number of nodes.*/
```

```
  NODE *t;              /* The tour pointer.*/
  int c;                /* The vehicles' capacity.*/
{
  NODE *np;             /* The index for the node pointer.*/
  NODE *lastnp;         /* The index for pointer to the last node.*/
  long infeas_ld = 0;   /* The total load infeasibility of the tour.*/
  long infeas_tw = 0;   /* The total TW infeasibility of the tour.*/
  lastnp = &t[n-1];
  for (np = &t[1]; np <= lastnp ; ++np) {
    infeas_tw += max(0, np->arr - np->l);
    if (np->type == 2) infeas_ld += max(0, np->load - c);
  }
  penptr->tw = infeas_tw;
  penptr->ld = infeas_ld;
  return;
}

/* A function to perform a swap move. It swaps two node structures in the specified tour.*/
void swap_node(i, j, t)
  int i, j;         /* Indices of the nodes to be swapped.*/
  NODE *t;          /* Structure for the tour to be swapped.*/
{
  NODE x;           /* Temporary variable for the SWAP macro.*/
  int z;            /* Temporary variable for the SWAP macro */

  SWAP(t[i].vehicle, t[j].vehicle, z);
  SWAP(t[i], t[j], x);
} /* end of swap node function */

/* A function to perform "depth" number sequence of swaps and returns a pointer to the resulting
tour structure vector.*/
struct node * insert(is, t, locate, depth, n)
  int is;       /* The node to be inserted.*/
  NODE *t; /* The current tour pointer.*/
  int depth;  /* The depth of the insertion (depth> 0) =>later; (depth<0) => earlier in the tour.*/
  int n;        /* The number of nodes in the tour. */
  int *locate;  /* locator array */
{
  register int i, j;   /* Indices for counting.*/
  NODE x;              /* Temporary variable for the SWAP macro.*/
  int z;               /* Temporary variable for the SWAP macro */
  NODE *t_t;           /* The structure for the new "inserted" tour.*/

  t_t = (struct node *)calloc(n, sizeof(NODE));
  for (i=0; i< n; ++i) t_t[i] = t[i];
  if (depth > 0)  {
    for (j=0; j < depth; ++j) {
```

```
        if (t_t[is+j].type == 2) {
          t_t[is+j+1].vehicle = t_t[is+j].vehicle;
          SWAP(t_t[is+j], t_t[is+j+1], x);
          SWAP(locate[t_t[is+j].id], locate[t_t[is+j+1].id],z);
        }
        else if (t_t[is+j].type == 1 && t_t[is+j+1].type == 2) {
          t_t[is+j].vehicle = t_t[is+j+2].vehicle;
          SWAP(t_t[is+j], t_t[is+j+1], x);
          SWAP(locate[t_t[is+j].id], locate[t_t[is+j+1].id],z);
        }
        else {
          SWAP(t_t[is+j], t_t[is+j+1], x);
          SWAP(locate[t_t[is+j].id], locate[t_t[is+j+1].id],z);
        }
      } /*end for */
    } /* end if */
    else {
      for (j= 0; j > depth; --j) {
        if (t_t[is+j].type ==2 ) {
          t_t[is+j-1].vehicle = t_t[is+j+1].vehicle;
          SWAP(t_t[is+j], t_t[is+j-1], x);
          SWAP(locate[t_t[is+j].id], locate[t_t[is+j-1].id], z);
        }
        else if (t_t[is+j].type == 1 && t_t[is+j-1].type == 2) {
          t_t[is+j].vehicle = t_t[is+j-1].vehicle;
          SWAP(t_t[is+j], t_t[is+j-1], x);
          SWAP(locate[t_t[is+j].id], locate[t_t[is+j-1].id], z);
        }
        else {
          SWAP(t_t[is+j], t_t[is+j-1], x);
          SWAP(locate[t_t[is+j].id], locate[t_t[is+j-1].id], z);
        }
      } /*end for*/
    } /* end else */
    for (i=0; i< n; ++i) t[i] = t_t[i];
    free(t_t);
    return (t);
} /* end of the Insertion function. */

/* A Function to compute the sum of the waiting time. */
long sum_wait(n, t)
    int n;          /*The number of nodes in the tour.*/
    NODE *t;        /*The tour to be printed.*/
{
    NODE *np;       /*Node pointer indexes used to iterate.*/
    long sum;       /*The waiting time sum.*/
    sum = 0;
```

```
  for (np = &t[0]; np <= &t[n-1]; ++np) sum += np->wait;
  return sum;
}/*The end of the sum_wait function.*/

void build_tour(t, tt, nc, nn, cap)
  NODE *t;
  int **tt;
  int nc, nn, cap;
{
  NODE *oldtour;
  NODE *oldtour2;
  PENALTY part_pen;
  int *locx, *loctemp, *loctemp2;
  register int i, d, j, j2, d2;
  int startj;
  int depth = (nc-1)/2;
  int lastvehicle = nc;
  int current_vehicle, vehicle_id;   /* records which vehicle is last in tour */
  int ch_i, ch_spi, ch_d, esc_i, esc_d;
  long d_best;
  int nn_partial, nv_partial, nc_partial;
  long move_val, move_val2;
  long tour_length;
  locx = (int *)calloc( nn, sizeof(int) );
  loctemp = (int *)calloc( nn, sizeof(int) );
  loctemp2 = (int *)calloc( nn, sizeof(int) );
  oldtour = (struct node *) calloc(nn, sizeof(NODE));
  oldtour2 = (struct node *) calloc(nn, sizeof(NODE));

  ch_i = ch_spi = ch_d = esc_i = esc_d = 0;
  d_best = 99999;
/* Get first pairwise-POS scheduled on first route */

for (i=1; i<=nc; ++i) {
  if (t[i].succ != 0) {
    ch_i = t[i].id;              /* location of the predecessor */
    ch_spi = t[t[i].succ].id;    /* location of the successor */
    break;
  }
}
if (ch_i != 1) iinsert (ch_i,t,1-ch_i,nn);
if (ch_i < ch_spi) iinsert (ch_spi,t,2-ch_spi,nn);
else iinsert(ch_spi+1,t,1-ch_spi,nn);
 /* insert vehicle #1 in position #3 on the tour */
iinsert (nc,t,3-nc,nn);
length = tour_sched (1, 3, t, tt);
for ( i = 0; i < nn; ++i ) locx[t[i].id] = i;
```

```
/* number of nodes, vehicles and customers on partially constructed tour */
nn_partial = 3; nv_partial = 1; nc_partial = 2;

/* Determine the customer vehicle assignment */
for (i=nn_partial; i > 0; --i) {
   if (t[i].type == 2) {
      vehicle_id = t[i].id;
      t[i].vehicle = vehicle_id;
   }
   if (t[i].type == 1) t[i].vehicle = vehicle_id;
}
ch_i = ch_spi = ch_d = 0;
for (i=4; i< nn-2; ++i) {
   if (nc_partial == nc-1) break; /* search starts only with a predecessor node */
   if (t[i].type == 2 || t[i].pred != 0) continue;
   startj = 1; nextj:
   for (j=startj; j<nn-2 && t[j].vehicle <= lastvehicle; ++j) {
      if (t[j].type == 2) {
/* if you are attempting to insert before the last node on the route, the vehicle node, you must
check to see if you can insert the pairwise-POS. Checking to insert the predecessor (supply node)
before the vehicle (depot) node is not permitted. It is one of the inadmissable arcs. This checks
SPATIAL fit */
      if (t[j-1].dep + tt[t[j-1].id][t[i].id] + tt[t[i].id][t[i].succ] + tt[t[i].succ][t[j].id]
         <= t[j].l && t[i].e + tt[t[i].id][t[i].succ] + tt[t[i].succ][t[j].id] <= t[j].l) {
         for (j2=0; j2 < nn ; ++j2)  {
            oldtour[j2] = t[j2];
            loctemp[j2] = locx[j2];
         } /* end for j2 - setting up working tour */

/* Insert the predecessor */
      oldtour = insert(i, oldtour, loctemp, j-i, nn);
/* Compute an initial change in the tour - this change IGNORES precedence and coupling */
      move_val = move_delta(i, j-i, nn, t, oldtour, tt);
         for (d2=0; d2 < nn; ++d2) {
            oldtour2[d2] = oldtour[d2];
            loctemp2[d2] = loctemp[d2];
         } /* end for d2 */

         oldtour2 = insert(loctemp[t[i].succ], oldtour2, loctemp2, j+1+-loctemp[t[i].succ], nn);
         move_val2 = move_delta(loctemp[t[i].succ], j+1+-loctemp[t[i].succ], nn, oldtour,
            oldtour2, tt) + move_val;
/* This checks TEMPORAL feasibility */
         comp_parpens(&part_pen, nn_partial+2, oldtour2, cap);
      /* if not feasible, disregard */
         if (part_pen.tw == 0 && part_pen.ld == 0 && move_val2 < d_best) {
            d_best = move_val2;
            ch_i = i;
```

```
        ch_spi = j;
        ch_d = 0;
      }
      if (t[j].vehicle == lastvehicle) goto tourupdate;
/* end insertion search and update the tour */
      else {
        startj = j+1;
        goto nextj;
      } /* look to insert a pair on the next sequenced route */
    }
  } /* end insertion of the pairwise-POS */
/* insert BEFORE the j position if it does not violate strong TWs.  This checks SPATIAL fit */

  if (t[j-1].dep + tt[t[j-1].id][t[i].id] + tt[t[i].id][t[j].id] <= t[j].l
    && t[i].e + tt[t[i].id][t[j].id] <= t[j].l) {
    current_vehicle = t[j].vehicle;
    for (j2=0; j2 < nn ; ++j2)  {
      oldtour[j2] = t[j2];
      loctemp[j2] = locx[j2];
    } /* end for j2 - setting up working tour */

/* Insert the predecessor */
    oldtour = insert(i, oldtour, loctemp, j-i, nn);
/* Compute an initial change in the tour - this change IGNORES precedence and coupling */
    move_val = move_delta(i, j-i, nn, t, oldtour, tt);
    for (d=0; d<=depth && t[j+d].vehicle == current_vehicle; ++d) {
/* Start with d=0 so you can insert the successor immediately following the predecessor.  Need to
save the passing of the predecessor for oldtour in preparation for the move of the successor */
      for (d2=0; d2 < nn; ++d2) {
        oldtour2[d2] = oldtour[d2];
        loctemp2[d2] = loctemp[d2];
      } /* end for d2 */

      oldtour2 = insert(loctemp[t[i].succ], oldtour2, loctemp2,  j+1+d-loctemp[t[i].succ], nn);
      move_val2 = move_delta(loctemp[t[i].succ], j+1+d-loctemp[t[i].succ], nn, oldtour,
        oldtour2, tt) + move_val;
/* This checks TEMPORAL feasibility */
      comp_parpens(&part_pen, nn_partial+2, oldtour2, cap);
    /* if not feasible, disregard */
      if (part_pen.tw > 0 || part_pen.ld > 0) continue;
      if (move_val2 < d_best) {
        d_best = move_val2;
        ch_i = i;
        ch_spi = j;
        ch_d = d;
      }
    } /* end for d - locating successor */
```

```
    } /* end if feasible insertion for predecessor */
  } /* end for j */ tourupdate:

  if (ch_i != 0) {
    esc_i = ((ch_i < locx[t[ch_i].succ]) ? locx[t[ch_i].succ] : locx[t[ch_i].succ]+1);
    esc_d = ch_spi+1+ch_d-esc_i;
    t = insert(ch_i, t, locx, ch_spi-ch_i, nn);
    t = insert(esc_i, t, locx, esc_d, nn);
  /* number of nodes, vehicles and customers on partially constructed tour */
    nn_partial = nn_partial + 2;
    nc_partial = nc_partial + 2;
  }
  if (ch_i == 0) {  /* need to start a new route */
    ch_i = locx[t[i].id];
    ch_spi = locx[lastvehicle]+1;
    esc_i = ((ch_i < locx[t[ch_i].succ]) ? locx[t[ch_i].succ] : locx[t[ch_i].succ]+1);
    esc_d = ch_spi+1+-esc_i;
    t = insert(ch_i, t, locx, ch_spi-ch_i, nn);
    t = insert(esc_i, t, locx, esc_d, nn); /* Insert the next vehicle */
    t = insert(locx[lastvehicle+1], t, locx, ch_spi+2-locx[lastvehicle+1], nn);
    ++lastvehicle;
    nn_partial = nn_partial + 3;
    nc_partial = nc_partial + 2;
    ++nv_partial;
  } /* Determine the customer vehicle assignment */
  tour_length = tour_sched(1, nn_partial, t, tt);
  for (i=nn_partial; i > 0; --i) {
    if (t[i].type == 2) {
      vehicle_id = t[i].id;
      t[i].vehicle = vehicle_id;
    }
    if (t[i].type == 1) t[i].vehicle = vehicle_id;
  }
    d_best = 99999;
    ch_i = ch_spi = ch_d = esc_i = esc_d = 0;
    i = nn_partial;
   } /* end for i - search for predecessors */
    free(locx);
    free(loctemp);
    free(loctemp2);
    free(oldtour);
    free(oldtour2);
    return;
  }
  /* Performs "depth" number sequence of swaps and returns a pointer to the resulting tour
  structure vector.*/
  void iinsert (is, t, depth, n)
```

```
  int is;       /* The node to be inserted.*/
  NODE *t; /* The current tour pointer.*/
  int depth;  /* The depth of the insertion (depth> 0) => later; (depth<0) => earlier in the tour.*/
  int n;        /* The number of nodes in the tour.*/
{
  register int j;   /* Indices for counting.*/
  NODE x;        /* Temporary variable for the SWAP macro.*/
  int z;
  if (depth > 0)  {
    for (j=0; j < depth; ++j) {
/* && (t[is+j].succ != t[is+j+1].id || t[is+j].id != t[is+j+1].pred); ++j) { */
      SWAP(t[is+j].vehicle, t[is+j+1].vehicle, z);
      SWAP(t[is+j], t[is+j+1], x);
    }
  } /* endif */
  else {
    for (j=0; j > depth; --j) {
/* && (t[is+j].pred != t[is+j+1].id || t[is+j].id != t[is+j-1].succ); --j) { */
      SWAP(t[is+j].vehicle, t[is+j-1].vehicle, z);
      SWAP(t[is+j], t[is+j-1], x);
    }
  } /* end else */
  return ;
} /* end of the Insertion function. */
void main(void)
{
  int numpbms;        /*The number of different problems to solve.*/
  int gamma = 0;      /*The penalty for using an additional vehicle.*/
  int *cap;           /*The vector of vehicle capacities by problem.*/
  int i;              /*Index.*/
  int iters;          /*The desired number of iterations.*/
  float *soln;        /*The array of the solution parameters.*/
  FILE *ifp;          /*The pointer to the problem input file.*/
  FILE *ofp;          /*The pointer to the problem output file.*/
  FILE *ofpi;         /*The pointer to the individual output file.*/
  FILE *fpinit;       /*The pointer to the initial tour data.*/
  char fileloc[30];   /*The directory location of the data files.*/
  char filename[50]; /*The exact file name.*/
  char infile[21];    /*The character array name of the input file.*/
  char outfile[21];   /*The char array name of the output file.*/
  char *p;            /*Character pointer used to "clean up" filenames.*/
  char **pbm;         /*The problem name.*/
  double twpen;       /*The TW Penalty term.*/
  double startpen;    /*The starting penalty.*/
  double endpen;      /*The ending penalty.*/
  double penincr;     /*The penalty incremental value.*/
  double vpen = 100.0;   /*The penalty factor for vehicle overload.*/
```

```
/*From stdio, input the number of problems to be attempted and the number of iterations to be
performed on each problem.*/
   input_fn(fileloc);
   printf("\nHow many problems do you want to solve?\t");
   scanf("%d", &numpbms);
   pbm = (char **)calloc(numpbms, sizeof(char *));
   for (i=0; i<numpbms; ++i)
     pbm[i] = (char *)calloc(MAXPBM, sizeof(char));
   cap = (int *)calloc(numpbms, sizeof(int));
   printf("\nHow many iterations (per problem) do you want to use?\t");
   scanf("%d", &iters);
   startpen = 1;
   twpen = startpen;   /*Input the starting penalty.*/
   endpen = 1;
   penincr = 1;
   printf("\nInput the problem name and vehicle capacity for the problem.\n");
   for (i=0; i<numpbms; ++i)
     scanf("%s %d", pbm[i], &cap[i]);
   input_vpen(&vpen);

/*Open the individual problem output file.*/
   ofpi = fopen("wes.out", "a");
   fprintf(ofpi,"The results for the problems for %d iterations each are:\n", iters);
   fprintf(ofpi,"The vehicle overload penalty is: %6.1f.\n", vpen);
   fclose(ofpi);
/*Open the individual problem output file.*/
   fpinit = fopen("wbinit.out", "a");
   fprintf(fpinit,"The results for the problems for %d iterations each are:\n", iters);
   fprintf(fpinit,"The vehicle overload penalty is: %6.1f.\n", vpen);
   fclose(fpinit);

/* MAJOR LOOP */
   for(twpen; twpen <= endpen; twpen += penincr)  {

/*Print the initial data to the individual summary output file: vrpstart.iout.*/
   ofpi = fopen("wes.out", "a");
   fprintf(ofpi,"The penalty term is: %5.2f\n\n", twpen);
   fprintf(ofpi, "        FEAS  TVL      ITER    Best  TVL      TOT\n");
   fprintf(ofpi, " PROBLEM  Zc(T) TIME VEH  NO     Time  TIME  PEN  VEH
TIME\n\n");  /*Print the initial data to initial.out.*/
   fpinit = fopen("wbinit.out", "a");
   fprintf(fpinit,"The time windows penalty term is: %5.2f\n\n", twpen);
   fprintf(fpinit," PROBLEM  Zt(T)   VEH   Time\n\n");
   fclose(fpinit);
   for (i = 0; i < numpbms; ++i)  {
     strcpy(infile, pbm[i]);
         p = strrchr( infile, '\n' );   /* remove newline if found */
```

```
    if ( p != NULL ) *p = '\0';
    printf("%s\n", infile);
/*Copy directory location into the filename, and add the filename to it.*/
    strcpy(filename, fileloc);
    strcat(filename, infile);
    strcat(filename, ".dat");
    printf ("%s\n",filename);

/*Copy the input file to the output file and append "out" to the filename.*/
    strcpy(outfile, infile);
    strcat(outfile, ".out");
/*Open the output file for the individual problem.*/
    ofp = fopen(outfile, "a");
    fprintf(ofp,"%8s ", outfile);
    fclose(ofp);
    ofp = fopen(outfile, "a");
/*Print the problem name to initial.out and close it.*/
    fpinit = fopen("wbinit.out", "a");
    fprintf(fpinit,"%8s ", infile);
    if( (ifp = fopen(filename, "r")) == NULL)
      fprintf (fpinit," The file was not opened") ;
    else fprintf (fpinit," The file was opened") ;
    fclose(fpinit);

/* The tabu search routine. */
    soln = t_search(ifp, ofp, iters, twpen, gamma, vpen, cap[i],infile[21]);

/*Output the individual problem results to the summary file.*/
    fprintf(ofpi, "%8s %6.1f  %6.1f %3d %4d  %8.2f| %6.1f %5.1f
      %3d  %10.2f\n",infile, soln[0], soln[1], ((soln[7] > 0)?
      (int)soln[12]: (int)soln[13]), (int) soln[2], soln[6], soln[3],
      soln[4], (int) soln[13], soln[9]);
    printf( "        FEAS  TVL      ITER    Best  TVL   TOT\n");
    printf( " PROBLEM  Zc(T)  TIME  VEH  NO     Time  TIME
      PEN  VEH  TIME\n\n");
    printf("%8s %6.1f  %6.1f %3d %4d  %8.2f| %6.1f %5.1f  %3d
      %10.2f\n",infile, soln[0], soln[1], ((soln[7] > 0)? (int)soln[12]:
      (int)soln[13]), (int) soln[2], soln[6], soln[3], soln[4],
      (int) soln[13], soln[9]);
    fclose(ifp);
    fclose(ofp);
  }/*end for numpbms*/
  fprintf(ofpi,"\n\n");
  fclose(ofpi);
} /*end for pen*/
  printf("\a\nFINISHED!...The output is in the file named  wes.out,\n");
}/* end of main program */
```

```
void input_fn(char *fileloc)
{
   int i;
   char c, d;
   char f[40] ;
   char g[5];
   strcpy(f,"/u/wes/nanu/n");
   printf("\nThe Solomon problems have 25, 50, or 100 customer problems.\n");
   printf("Which problem set is required?\t");
   gets(g);
   strcat(f, g);
   strcat(f, "/");
   printf("\n\nThe file location is \t<%s>.\n", f);
   printf("Is this the correct location? <Y, N>\t");
   for (i=0; (c = getchar()) != '\n'; ++i) d = c;
   while (d != 'Y' && d != 'y') {
      printf("\nPlease input the name of the directory where your data files are located.\n");
      gets(f);
      printf("The file location is %s.\n", f);
      printf("Is this the correct location? <Y, N>\t");
      for (i=0; (c = getchar()) != '\n'; ++i) d = c;
   } /*end while*/
/*Copy the correct input filename to fileloc for the main program.*/
   for(i=0; (fileloc[i] = f[i]) != '\0'; ++i);
   return;
}/*end of input file name function.*/
```

Reference List

Atkinson, Ben J (1994), A Greedy Look-ahead Heuristic for Combinatorial Optimization: An Application to Vehicle Scheduling with Time Windows, *Journal of the Operational Research Society*, vol. 45, no. 6, pp. 673-684.

Baker, Edward K. and Joanne R. Schaffer (1986), Solution Improvement Heuristics for the Vehicle Routing and Scheduling Problem with Time Window Constraints, *American Journal of Mathematical and Management Sciences*, Vol. 6, Nos. 3 & 4, pp. 261-300.

Balakrishnan, Nagraj (1993), Simple Heuristics for the Vehicle Routing Problem with Soft Time Windows, *Journal of the Operational Research Society*, vol. 44, no. 3, pp. 279-287.

Barnes, J.W. and W.B. Carlton (1995), "Solving the Vehicle Routing Problem with Time Windows Using Reactive Tabu Search," presented at the Fall INFORMS Conference in New Orleans, Louisiana, October 31, 1995.

Battiti, R. and G. Tecchiolli (1994), The Reactive Tabu Search, *ORSA Journal of Computing*, vol. 6, no. 2, pp. 126-140.

Battiti, Roberto (1995), Reactive Search: Toward Self-Tuning Heuristics, *Keynote talk at Applied Decision Technologies*, 3-4 April 1995, Brunel, UK.

Baugh, John W. Jr (1995), Multiobjective Optimization of the Dial-a-Ride Problem Using Simulated Annealing, *Computing in Civil Engineering*, vol. 1, pp. 278-285.

Beaujon, George J. and Mark A. Turnquist (1991), Model for Fleet Sizing and Vehicle Allocation, *Transportation Science*, vol. 25, no. 1, pp. 19-45.

Bianco, L., A. Mingozzi, S. Ricciardelli and M. Spadoni (1994), Exact and Heuristic Procedures for the Traveling Salesman Problem with Precedence Constraints, Based on Dynamic Programming, *Infor*, vol. 32, no. 1, pp. 19-32.

Bodin, L, B. Golden, A. Assad and M. Ball (1983), Routing and Scheduling of Vehicles and Crews: The State of the Art, *Computers and Operations Research*, vol. 10, No. 2, pp. 62-211.

Bramel, J., C.L. Li and D. Simchi-Levi (1993), Probabilistic Analysis of a Vehicle Routing Problem with Time Windows, *American Journal of Mathematical and Management Sciences*, vol. 13, nos. 3 & 4, pp. 267-322.

Buyang, Cao and Gotz Uebe (1995), Solving Transportation Problems with Nonlinear Side Constraints with Tabu Search, *Computers and Operations Research*, vol. 22, no. 6, pp. 593-603.

Carlton, W.B. (1995), *A Tabu Search Approach to the General Vehicle Routing Problem*, Ph. D. Dissertation, Department of Mechanical Engineering, University of Texas at Austin.

Carlton, W.B. and J.W. Barnes (1995), A Note on Hashing Functions and Tabu Search Algorithms, *European Journal of Operational Research*, vol. 95, pp. 237-239 .

Chao, I.M., Bruce Golden and Edward Wasil (1993), A New Heuristic for the Multi-Depot Vehicle Routing Problem that Improves Upon Best-Known Solutions, *American Journal of Mathematical and Management Sciences*, vol. 13, nos. 3 & 4, pp. 371-406.

Christofides, N., A. Mignozzie and P. Toth (1981), State-Space Relaxation Procedures for the Computation of Bounds to Routing Problems, *Networks*, vol. 11, pp. 145-164.

Clarke, G. and J.W. Wright (1964), Scheduling of Vehicles from a Central Depot to a Number of Delivery Points, *Operations Research*, vol. 12, p. 568-581.

Dantzig, G.B. and K.H. Ramser (1959), The Truck Dispatching Problem, *Operations Research*, vol. 12, pp. 80-91.

Derigs, U. and G. Grabenbauer (1993), Intime - A New Heuristic Approach to the Vehicle Routing Problem with Time Windows with a Bakery Fleet Case, *American Journal of Mathematical and Management Sciences*, vol. 13, nos. 3 & 4, pp. 249-266.

Desrochers, M., J. Desrosiers and M. M. Solomon (1992), A New Optimization Algorithm for the Vehicle Routing Problem with Time Windows, *Operations Research*, vol. 40, pp. 342-354 .

Desrochers, M. and T.W. Verhoog (1991), New heuristic for the Fleet Size and Mix Vehicle Routing Problem, *Computers and Operations Research*, vol. 18, no. 3, pp. 263-274.

Desrochers, M, J.K. Lenstra and M.W.P Savelsbergh (1990), A Classification Scheme for Vehicle Routing, *European Journal of Operational Research*, vol. 46, pp. 322-332.

Desrochers, M, J.K. Lenstra, M.W.P. Savelsbergh and F. Soumis (1988), Vehicle Routing with Time Windows: Optimization and Approximation, *Vehicle Routing: Methods and Studies*, Elsevier, New York.

Desrochers, Martin and Francois Soumis (1988), A Generalized Permanent Labeling Algorithm for the Shortest Path Problem with Time Windows, *INFOR*, vol. 26, no. 3, pp. 191-212.

Desrosiers, J., Y. Dumas, M. Solomon and F. Soumis (1995), Time Constrained Routing and Scheduling, Handbooks on Operational Research and Management Science, vol. 8, *Network Routing*, M.O. Ball, T.L. Magnanti, C.L. Monma and G.L. Nemhauser eds., Amsterdam, North-Holland Press, pp. 35-139.

Desrosiers, J., G. LaPorte, M. Sauve, F. Soumis and S. Taillefer (1988), Vehicle Routing with Full Loads, *Computers and Operations Research*, vol. 15, no. 3, pp. 219-226.

Desrosiers, J., F. Soumis and M. Desrochers (1984), Routing with Time Windows by Column Generation, *Networks*, vol. 14, pp. 545-565.

Dror, Moshe (1994), Note on the Complexity of the Shortest Path Models for Column Generation in VRPTW, *Operations Research*, vol. 42, no. 5, pp. 977-978.

Dumas, Y., J. Desrosiers, E. Gelinas and M. Solomon (1995), An Optimal Algorithm for the Traveling Salesman Problem with Time Windows, *Operations Research*, vol. 43, no. 2, pp. 367-371.

Dumas, Y., J. Desrosiers and F. Soumis (1991), The Pickup and Delivery Problem with Time Windows, *European Journal of Operational Research*, vol. 54, pp. 7-22.

Fisher, Marshall, K. Jornsten and O. Madsen (1997), Vehicle Routing with Time Windows: Two Optimization Algorithms, *Operations Research*, vol. 45, no. 3, pp. 488-492.

Fisher, Marshall (1995), Vehicle Routing, Handbooks on Operational Research and Management Sciences, vol. 8, *Network Routing*, M.O. Ball, T.L. Magnanti, C.L. Monma and G.L. Nemhauser eds., Amsterdam, North-Holland Press, pp. 1-33.

Fisher, M.L. and R. Jaikumar (1981), A Generalized Assignment Heuristic for Vehicle Routing, *Networks*, vol. 11, pp. 109-124.

Frizzell, P.W. and J.W. Griffin (1995), The Split Delivery Vehicle Scheduling Problem with Time Windows and Grid Network Distances, *Computers and Operations Research*, vol. 22, no. 6, pp. 655-667.

Garcia, B.L., J. Potvin and J. Rousseau (1944), A Parallel Implementation of the Tabu Search Heuristic for Vehicle Routing Problems with Time Window Constraints, *Computers and Operations Research*, vol. 21, no. 9, pp. 1025-1033.

Gendreau, M., A. Hertz and G. LaPorte (1994), A Tabu Search Heuristic for the Vehicle Routing Problem, *Management Science*, vol.. 40, no. 10, pp. 1276-1290.

Glover, Fred (1996), Tabu Search and Adaptive Memory Programming Advances, Applications and Challenges, to appear in *Interfaces in Computer Science and Operations Research.*

Glover, Fred (1996), Reflections and Research Possibilities, e-mail to Dr. Wes Barnes.

Glover, Fred (1995), "Tabu Search Fundamentals and Uses." Graduate School of Business, University of Colorado, condensed version published in *Mathematical Programming: State of the Art*, Birge and Murty, eds., pp. 64-92.

Glover, Fred (1990), Tabu Search: A Tutorial, *Interfaces*, vol. 20, no. 4, pp. 74-94.

Glover, Fred (1989), Tabu Search - Part I, *ORSA Journal of Computing*, vol. 1, no. 3, pp. 190-206.

Goetschalckx, M. and C. Jacobs-Blechs (1989), The Vehicle Routing Problem with Backhauls, *European Journal of Operational Research*, vol. 42, pp. 39-51.

Golden, Bruce L. and Arjang A. Assad (1986), Vehicle Routing with Time-Window Constraints, *American Journal of Mathematical and Management Sciences*, nos. 3 & 4, pp. 251-260.

Healy, Patrick and Robert Moll (1995), A New Extension of Local Search Applied to the Dial-a-Ride Problem, *European Journal of Operational Research*, vol. 83, pp. 83-104.

Horowitz, E., S. Sahni an S. Anderson-Freed (1993), *Fundamentals of Data Structures in C*, New York, W.H. Freeman and Company.

Jarvis, J.J. and O. Kirca (1985), Pick-up and Delivery Problem: Models and Single Vehicle Exact Procedures, PDRC Report Series 84-12.

Jensen, Paul and J.W. Barnes (1980), *Network Flow Programming*, New York, NY, John Wiley& Sons.

Kalantari, B., A.V. Hill and S.R. Arora (1985), An Algorithm for the Traveling Salesman Problem with Pickup and Delivery Customers, *European Journal of Operational Research*, vol. 22, pp. 377-386.

Kantor, Marisa G. and Moshe B. Rosenwein (1992), The Orienteering Problem with Time Windows, *Journal of the Operational Research Society*, vol. 43, no. 6, pp. 629-635.

Kelly, J, B. Golden and A. Assad (1993), Large-scale Controlled Rounding using Tabu Search with Strategic Oscillation, *Annals of Operations Research*, vol. 41, pp. 69-84.

Kohl, Niklas (1995), *Exact Method for Time Constrainted Routing and Scheduling Problems,*Ph.D. Dissertation, Department of Mathematics University of Copenhagen, Denmark..

Kohl, Niklas and Oli B.G. Madsen (1997), An Optimization Algorithm for the Vehicle Routing Problem with Time Windows Based on Lagrangian Relaxation, *Operations Research*, vol. 45, no. 3, pp. 395-406.

Kolen, A.W.J., A.H.G. Rinooy Kan and H.W.J.M. Trienekens, (1987), Vehicle Routing with Time Windows, *Operations Research*, vol. 35, no. 2, pp. 266-273.

Kontoravdis, G. and J. Bard (1993), Improved Heuristics for the Vehicle Routing Problem with Time Windows, Working Paper, Operations Research Group, Department of Mechanical Engineering, The University of Texas at Austin, Austin, TX.

Koskosidis, Y., W. Powell and M. Solomon (1992), An Optimization-Based Heuristic for Vehicle Routing and Scheduling with Soft Time Window Constraints, *Transportation Science*, vol. 26, no. 2, p. 69-85.

Koskosidis, Y.A. and W.B. Powell (1990), Application of Optimization Based Models on Vehicle Routing, *Journal of Business Logistics*, vol. 11, pp. 101-127.

Landeghem, H.R.G. (1988), A Bi-Criteria Heuristic for the Vehicle Routing Problem with Time Windows, *European Journal of Operational Research*, vol. 36, pp. 217-226.

LaPorte, Gilbert (1992), The Vehicle Routing Problem: An Overview of Exact and Approximate Algorithms, *European Journal of Operational Research*, vol. 59, pp. 345-358.

Mingozzi, A., L. Bianco and S. Ricciardelli (1997), Dynamic Programming Strategies for the Traveling Salesman Problem with Time Window and Precedence Constraints, *Operations Research*, vol. 45, no. 3, pp. 365-376.

Or, I. (1976), *Traveling Salesman-Type Combinatorial Optimization Problems and Their Relation to the Logistics of Blood Banking*, Ph.D. dissertation, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, Illinois.

Osman, I.H. (1993), Metastrategy Simulated Annealing and Tabu Search Algorithms for the Vehicle Routing Problem, *Annals of Operations Research*, vol. 41, pp. 421-451.

Potvin, J., T. Kervahut, B. Garcia and J. Rousseau (1993), A Tabu Search Heuristic for the Vehicle Routing Problem with Time Windows, Working Paper, Centre de Recherche sur les Transports, Universite de Montreal, Montreal, CA.

Potvin, J.Y. and J.M. Rousseau (1993), A Parallel Route Building Algorithm for the Vehicle Routing and Scheduling Problem with Time Windows, *European Journal of Operational Research*, vol. 66, pp. 331-340.

Potvin, Jean-Yves and Jean-Marc Rousseau (1995), An Exchange Heuristic for Routing Problems with Time Windows, *Journal of the Operational Research Society*, vol. 46, pp. 1433-1446.

Psaraftis, H. (1980), A Dynamic Programming Solution to the Single Vehicle Many-to-Many Immediate Request Dial-a-Ride Problem, *Transportation Science*, vol. 14, pp. 130-154.

Psaraftis, H. (1983), An Exact Algorithm for the Single Vehicle Many-to-Many Dial-A-Ride Problem with Time Windows, *Transportation Science*, vol. 17, No. 3, pp. 351-361.

Psaraftis, H. (1986), Scheduling Large-Scale Advance-Request Dial-a-Ride Systems, *American Journal of Mathematical and Management Sciences*, vol. 6, Nos. 3 & 4, pp. 327-367.

Punnen, Abraham P. and Y.P Aneja (1995), A Tabu Search Algorithm for the Resource-Constrained Assignment Problem, *Journal of the Operational Research Society*, vol. 46, pp. 214-220.

Raghavendra A., T.S. Krishnakumar, R. Muralidhar and D. Sarvanan (1992), A Practical Heuristic for a Large Scale Vehicle Routing Problem, *European Journal of Operational Research*, vol. 57, pp. 32-38.

Reeves, Colin (1993), Improving the Efficiency of Tabu Search for Machine Sequencing Problems, *Journal of the Operational Research Society*, vol. 44, no. 4, pp. 375-382.

Rego, Cesar and C. Roucairol (1995), Using Tabu Search for Solving a Dynamic Multi-Terminal Truck Dispatching Problem, *European Journal of Operational Research*, vol. 83, pp. 411-429.

Rochat, Y and F. Semet (1994), A Tabu Search Approach for Delivering Pet Food and Flour in Switzerland, *Journal of the Operational Research Society*, vol. 45, no. 11, pp. 1233-1245.

Ruland, Scott (1995), *Polyhedral Solution to the Pickup and Delivery Problem*, Ph.D. Dissertation, Washington University Sever Institute of Technology.

Savelsbergh, M.W.P. (1992), The Vehicle Routing Problem with Time Windows: Minimizing Route Duration, *ORSA Journal on Computing*, vol. 4, pp. 146-154.

Savelsbergh, M.W.P. (1990), An efficient implementation of local search algorithms for constrained routing problems, *European Journal of Operations Research*, vol. 47, pp. 75-85.

Semet, F. and E. Taillard (1993), Solving Real-Life Vehicle Routing Problems Efficiently Using Tabu Search, *Annals of Operations Research*, vol. 41, pp. 469-488.

Sexton, Thomas R. and Lawrence D. Bodin (1985a). Optimizing Single Vehicle Many-to-Many Operations with Desired Delivery Times: I. Scheduling, *Transportation Science*, vol. 19, no. 4, pp. 378-410.

Sexton, Thomas R. and Lawrence D. Bodin (1985b). Optimizing Single Vehicle Many-to-Many Operations with Desired Delivery Times: II. Routing, *Transportation Science*, vol. 19, no. 4, pp. 411-435.

Sexton, Thomas R. and Young-Myung Choi (1986), Pickup and Delivery of Partial Loads with "Soft" Time Windows, *American Journal of Mathematical and Management Sciences*, vol. 6, nos. 3 & 4, pp. 369-398.

Sharma, R.R.K. (1995), Modeling a Railway Freight Transport System, *Asia-Pacific Journal of Operations Research*, vol. 12, pp. 17-36.

Solomon, M.M., E.K. Baker and J.R. Schaffer (1988), Vehicle Routing and Scheduling Problems with Time Window Constraints: Efficient Implementations of Solution Improvement Procedures, *Vehicle Routing: Methods and Studies*, pp. 85-105, North-Holland, Amsterdam.

Solomon, M., and J. Desrosiers (1988), Survey Paper: Time Window Constrained Routing and Scheduling Problems, *Transportation Science*, vol. 22, no. 1, pp. 1-13.

Solomon, Marius M. (1986), On the Worst-Case Performance of Some Heuristics for the Vehicle Routing and Scheduling Problem with Time Window Constraints, *Networks*, vol. 16, pp. 161-174.

Solomon, M. (1987), Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints, *Operations Research*, vol. 35, no. 2, pp. 254-265.

Stewart, W.R., J.P. Kelly and M. Laguna (1993), Solving Vehicle Routing Problems Using Generalized Assignments and Tabu Search, Working Paper, Graduate School of Business Administration, The College of William and Mary, Williamsburg, VA.

Taillard, E., P. Badeau, M. Gendreau, F. Guertin and J. Potvin (1996), A New Neighborhood Structure for the Vehicle Routing Problem with Time Windows, working paper to appear in *Transportation Science*.

Timlin, M.T. Fiala and W.R. Pulleyblank (1992), Precedence Constrained Routing and Helicopter Scheduling: Heuristic Design, *Interfaces*, vol. 22, no. 3, pp. 100-111.

Thompson, P.M. and H. Psaraftis (1993), Cyclic Transfer Algorithms for Multivehicle Routing and Scheduling Problems, *Operations Research*, vol. 41, no. 5, pp. 935-946.

Van der Bruggen, L.J.J., J.K. Lenstra, and P.C. Schuur (1993), Variable-Depth Search for the Single Vehicle Pickup and Delivery Problem with Time Windows, *Transportation Science*, vol. 27, no. 3, pp. 298-311.

Vita

William Paul Nanry was born in East Patchogue, New York, on June 17, 1957, the son of Muriel Theresa Nanry and James Joseph Nanry, Sr. After completing his work at Connetquot High School, Bohemia, New York, in June of 1975, he entered the United States Military Academy (USMA) in West Point, New York. He received the degree of Bachelor of Science from USMA and was commissioned a second lieutenant in the United States Army Corps of Engineers in June 1979. During the following years he was employed in numerous positions in the Army to include serving as a commander of a 165-man combat engineer company at Fort Devens, Massachusetts. Due to his exemplary service, he was afforded the opportunity to attend the University of Texas at Austin and earned a Master of Arts in Mathematics in May of 1989. While serving as an Assistant Professor in the Mathematics Department at USMA, he switched his military specialty to Operations Research. Lieutenant Colonel Nanry initiated his studies toward the Doctor of Philosophy degree in Operations Research upon enrolling at the University of Texas at Austin in August of 1994.

Permanent Address: 4301 Candlestick Court, Montclair, VA 22026.

This report was typed by the author.

PLEASE CHECK THE APPROPRIATE BLOCK BELOW:

-AQ #_____

☐ _____ copies are being forwarded. Indicate whether Statement A, B, C, D, E, F, or X applies.

☑ DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE: DISTRIBUTION IS UNLIMITED

☐ DISTRIBUTION STATEMENT B:
DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES ONLY; (Indicate Reason and Date). OTHER REQUESTS FOR THIS DOCUMENT SHALL BE REFERRED TO (Indicate Controlling DoD Office).

☐ DISTRIBUTION STATEMENT C:
DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES AND THEIR CONTRACTORS; (Indicate Reason and Date). OTHER REQUESTS FOR THIS DOCUMENT SHALL BE REFERRED TO (Indicate Controlling DoD Office).

☐ DISTRIBUTION STATEMENT D:
DISTRIBUTION AUTHORIZED TO DoD AND U.S. DoD CONTRACTORS ONLY; (Indicate Reason and Date). OTHER REQUESTS SHALL BE REFERRED TO (Indicate Controlling DoD Office).

☐ DISTRIBUTION STATEMENT E:
DISTRIBUTION AUTHORIZED TO DoD COMPONENTS ONLY; (Indicate Reason and Date). OTHER REQUESTS SHALL BE REFERRED TO (Indicate Controlling DoD Office).

☐ DISTRIBUTION STATEMENT F:
FURTHER DISSEMINATION ONLY AS DIRECTED BY (Indicate Controlling DoD Office and Date) or HIGHER DoD AUTHORITY.

☐ DISTRIBUTION STATEMENT X:
DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES AND PRIVATE INDIVIDUALS OR ENTERPRISES ELIGIBLE TO OBTAIN EXPORT-CONTROLLED TECHNICAL DATA IN ACCORDANCE WITH DoD DIRECTIVE 5230.25, WITHHOLDING OF UNCLASSIFIED TECHNICAL DATA FROM PUBLIC DISCLOSURE, 6 Nov 1984 (Indicate date of determination). CONTROLLING DoD OFFICE IS (Indicate Controlling DoD Office).

☐ This document was previously forwarded to DTIC on _____ (date) and the AD number is _____.

☐ In accordance with provisions of DoD instructions, the document requested is not supplied because:

☐ It will be published at a later date. (Enter approximate date, if known).

☐ Other. (Give Reason)

DoD Directive 5230.24, "Distribution Statements on Technical Documents," 18 Mar 87, contains seven distribution statements, as described briefly above. Technical Documents must be assigned distribution statements.

X WILLIAM P. NANRY
Print or Type Name

X 7/16/98

X William P. Nanry