DYNAMIC UNMANNED AERIAL VEHICLE (UAV)
ROUTING WITH A JAVA-ENCODED
REACTIVE TABU SEARCH METAHEURISTIC

THESIS

Kevin P. O'Rourke, Captain, USAF

AFIT/GOA/ENS/99M-06

Approved for public release; distribution unlimited.
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the US Government.
DYNAMIC UNMANNED AERIAL VEHICLE (UAV) ROUTING WITH A JAVA-ENCODED REACTIVE TABU SEARCH METAHEURISTIC

THESIS

Presented to the Faculty of the Graduate School of Engineering
Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Operational Analysis

Kevin P. O’Rourke, B.S.
Captain, USAF

March 1999

Approved for public release; distribution unlimited
THESIS APPROVAL

NAME: Kevin P. O'Rourke, Captain, USAF     CLASS: GOA-99M

THESIS TITLE: Dynamic Unmanned Aerial Vehicle (UAV)
Routing with a Java-Encoded Reactive Tabu Search Metaheuristic

DEFENSE DATE: 2 March 1999

<table>
<thead>
<tr>
<th>COMMITTEE:</th>
<th>NAME/TITLE/DEPARTMENT</th>
<th>SIGNATURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advisor</td>
<td>T. Glenn Bailey, Lieutenant Colonel, USAF Assistant Professor of Operations Research Department of Operational Sciences Air Force Institute of Technology</td>
<td></td>
</tr>
<tr>
<td>Reader</td>
<td>Raymond R. Hill, Major, USAF Assistant Professor of Operations Research Department of Operational Sciences Air Force Institute of Technology</td>
<td></td>
</tr>
<tr>
<td>Reader</td>
<td>William B. Carlton, Lieutenant Colonel (P), USA Adjunct Assistant Professor of Operations Research Department of Operational Sciences Air Force Institute of Technology</td>
<td></td>
</tr>
</tbody>
</table>
Acknowledgments

I would like to extend my sincere thanks to those people who both helped with my thesis and made my time here at AFIT a success. Thanks to Lt Col Glenn Bailey, my thesis advisor, for providing guidance while giving me the latitude to explore ideas on my own. Thanks to LTC (P) Carlton for his tremendous initial work that provided the reactive tabu search and for his comments and feedback as a reader. Thanks to Major Ray Hill for his feedback and persuasive steering.

Thanks to Major Dave Ryer, who shared in the trials and tribulations of bringing this monster to life. I am glad that I helped our efforts, and certainly appreciated his work during the times that our thesis endeavors were on parallel tracks. I appreciate all the kind hospitality of people who helped after my knee surgery (as if graduate school wasn’t hard enough on its own)—Major Sam and Mary Szvetecz, who opened their home to provide a much needed change of scenery, as well as Dan Franzen and Lance Hrivnak who chauffeured me while I couldn’t drive. Thanks to all of my instructors and classmates for giving me the opportunity to learn something from each and every one of you.

I thank my friends here, both new and old (you all know who you are) for the support in the hard times and the necessary diversions to keep me sane. Thanks to my family for always being there, and for your 30 years of instruction on the lessons of life.

Kevin P. O’Rourke
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgments</td>
<td>iii</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>iv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
<tr>
<td>Notation</td>
<td>viii</td>
</tr>
<tr>
<td>Abstract</td>
<td>x</td>
</tr>
<tr>
<td>Chapter 1</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Reactive Tabu Search for the Vehicle Routing Problem with Time Windows</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1 Objective Function</td>
<td>10</td>
</tr>
<tr>
<td>2.2.2 Penalized Objective Function</td>
<td>11</td>
</tr>
<tr>
<td>2.2.3 Adjusting Reactive Penalty Coefficients</td>
<td>12</td>
</tr>
<tr>
<td>2.2.4 Initial Solution</td>
<td>13</td>
</tr>
<tr>
<td>2.2.5 Neighborhood Structure</td>
<td>14</td>
</tr>
<tr>
<td>2.2.6 Tabu Moves</td>
<td>15</td>
</tr>
<tr>
<td>2.2.7 Adjusting Tabu Length</td>
<td>16</td>
</tr>
<tr>
<td>2.2.8 Aspiration and Escape Functions</td>
<td>18</td>
</tr>
<tr>
<td>2.2.9 Move Evaluation and Selection</td>
<td>18</td>
</tr>
<tr>
<td>2.2.10 Heuristic Description</td>
<td>18</td>
</tr>
<tr>
<td>2.2.11 Computational Complexity</td>
<td>19</td>
</tr>
<tr>
<td>2.3 Reactive Tabu Search for Dynamic Unmanned Aerial Vehicle Routing</td>
<td>20</td>
</tr>
<tr>
<td>2.3.1 Operational Parameters</td>
<td>21</td>
</tr>
<tr>
<td>2.3.2 Geographic Coordinates</td>
<td>21</td>
</tr>
<tr>
<td>2.3.3 Wind Effects on Ground Speed</td>
<td>22</td>
</tr>
<tr>
<td>2.3.4 Numerical Formatting</td>
<td>23</td>
</tr>
<tr>
<td>2.3.5 Objective Function Modifications</td>
<td>24</td>
</tr>
<tr>
<td>2.3.6 Dynamic Mission Requirements</td>
<td>25</td>
</tr>
<tr>
<td>2.3.7 Optimizing Use of Altitude-Based Wind Tiers</td>
<td>25</td>
</tr>
<tr>
<td>2.3.8 Random Service Times</td>
<td>27</td>
</tr>
<tr>
<td>2.3.9 Emerging Targets</td>
<td>27</td>
</tr>
</tbody>
</table>
2.3.10 Locked and Forbidden Routes .......................................................... 28
2.4 Computational Results ......................................................................... 29
  2.4.1 General Results ............................................................................. 29
  2.4.2 UAV Results ................................................................................ 36
2.5 Conclusions ......................................................................................... 37

Chapter 3 .................................................................................................... 43
  3.1 Heuristic Modifications ....................................................................... 43
  3.2 UAV Related Modifications ................................................................. 44
  3.3 Java Code Modifications ..................................................................... 44

Bibliography ............................................................................................... 46

Appendix A. Extended Problem Formulation ............................................. 49
  A.1 Traveling Salesman Problem (TSP) ..................................................... 49
  A.2 Multiple Traveling Salesman Problem (MTSP) .................................. 51
  A.3 Vehicle Routing Problem (VRP) ......................................................... 53
  A.4 Multiple Depot Vehicle Routing Problem (MDVRP) ......................... 55

Appendix B. Tabu Search vs. Other Heuristics—TSP Example .................. 57

Appendix C. Javadoc Listing ...................................................................... 58

Appendix D. Additional References .......................................................... 116

Vita .............................................................................................................. 118
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Disjunctive Graph Notation</td>
<td>9</td>
</tr>
<tr>
<td>2. Initial Tour Sequence</td>
<td>10</td>
</tr>
<tr>
<td>3. Adjacent 3-Opt Swap Move</td>
<td>14</td>
</tr>
<tr>
<td>4. Redundant Tours</td>
<td>17</td>
</tr>
<tr>
<td>5. Distance and Bearing Geometry (Spherical Triangle)</td>
<td>22</td>
</tr>
<tr>
<td>6. Headwind and Tailwind Ground Speed Adjustment</td>
<td>23</td>
</tr>
<tr>
<td>7. Typical Winds Aloft Profile</td>
<td>26</td>
</tr>
<tr>
<td>8. Forbidden Route Example</td>
<td>29</td>
</tr>
<tr>
<td>9. Bosnia Scenario Target Locations</td>
<td>39</td>
</tr>
<tr>
<td>10. Bosnia Optimized Tour Route</td>
<td>41</td>
</tr>
<tr>
<td>11. Temporal Route Plot</td>
<td>42</td>
</tr>
</tbody>
</table>
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Solomon mTSPTW Computational Results (25 Customers)</td>
<td>30</td>
</tr>
<tr>
<td>2. Solomon mTSPTW Computational Results (50 Customers)</td>
<td>31</td>
</tr>
<tr>
<td>3. Solomon mTSPTW Computational Results (100 Customers)</td>
<td>32</td>
</tr>
<tr>
<td>4. Solomon VRPTW Computational Results (25 Customers)</td>
<td>33</td>
</tr>
<tr>
<td>5. Solomon VRPTW Computational Results (50 Customers)</td>
<td>34</td>
</tr>
<tr>
<td>6. Solomon VRPTW Computational Results (100 Customers)</td>
<td>35</td>
</tr>
<tr>
<td>7. Wind Data</td>
<td>36</td>
</tr>
<tr>
<td>8. Bosnia Data Set</td>
<td>38</td>
</tr>
<tr>
<td>9. Bosnia Tour Sequence</td>
<td>40</td>
</tr>
</tbody>
</table>
Notation

The following symbols appear in the main body of the paper and are defined as listed. For the sake of clarity, symbols appearing only in the appendices are defined when introduced.

\[ a_i \] = arrival time at node \( i \)
\[ A \] = arc set
\[ A \] = line segment, wind adjustment triangle
\[ AS \] = air speed
\[ B \] = line segment, wind adjustment triangle
\[ c_{ij} \] = cost (travel time) from node \( i \) to \( j \)
\[ C \] = line segment, wind adjustment triangle
\[ C_i \] = excess vehicle capacity
\[ d \] = insertion move depth
\[ d_i \] = departure time from node \( i \)
\[ d_{ij} \] = great circle distance between locations \( i \) and \( j \)
\[ D \] = maximum tour-length duration
\[ D_i \] = excess route duration
\[ e_i \] = earliest begin service time of node \( i \)
\[ G \] = graph set
\[ GS \] = ground speed
\[ H \] = intermediate heading angle
\[ h_{ijk} \] = travel altitude \( k \) between locations \( i \) and \( j \)
\[ i \] = insertion move position
\[ k \] = altitude band
\[ k \] = iteration
\[ \ell_i \] = latest begin service time of node \( i \)
\[ L \] = latitude
\[ LD \] = load overage violations
\[ m \] = number of vehicles
\[ n \] = number of customers
\[ q_i \] = non-negative customer demand (quantity) of node \( i \)
\[ Q \] = vehicle capacity
\[ s_i \] = customer service time of node \( i \)
\[ s_{\text{max}}(i) \] = maximum stochastic service time for location \( i \)
\[ s_{\text{min}}(i) \] = minimum stochastic service time for location \( i \)
\[ S_i \] = stochastic customer service time for location \( i \)
\[ t \] = arbitrary time within time window
\[ t_{ij} \] = travel time from node \( i \) to \( j \)
\[ t_{LD} \] = number of load infeasible solutions in the previous ten iterations
\[ t_{RW} \] = number of time window infeasible solutions in the previous ten iterations
\[ TW \] = time window violations
\( t_{hv}(t) \) = tour hashing value

\( n_0 \) = initial depot node (TSP, VRP)

\( n_i \) = additional nodes (TSP, VRP)

\( V \) = vertex set

\( w_i \) = wait time to commence at node \( i \)

\( WS \) = wind speed

\( x_{ij} \) = indicator variable denoting arc from node \( i \) to \( j \) is included in the tour

\( Z_f(t) \) = feasible objective function

\( Z(t) \) = penalized objective function

\( Z'_f(t) \) = feasible UAV objective function

\( Z'(t) \) = penalized UAV objective function

\( \Theta_{ij} \) = bearing from location \( i \) to \( j \)

\( \Theta_{WS} \) = wind bearing

\( \Psi_{ij} \) = random weight for arc \( i, j \)

\( \delta \) = course correction angle

\( \lambda \) = longitude

\( \theta \) = tabu list length

\( \rho \) = generic penalty scaling factor

\( \rho_{LD} \) = load capacity penalty scaling factor

\( \rho_{TW} \) = time window penalty scaling factor

\( \tau \) = tour position
Abstract

In this paper we consider the dynamic routing of unmanned aerial vehicles (UAVs) currently in operational use with the US Air Force. Dynamic vehicle routing problems (VRP) have always been challenging, and the airborne version of the VRP adds dimensions and difficulties not present in typical ground-based applications. Previous UAV routing work has focused on primarily on static, pre-planned situations; however, scheduling military operations, which are often ad-hoc, drives the need for a dynamic route solver that can respond to rapidly evolving problem constraints. With these considerations in mind, we examine the use of a Java-encoded metaheuristic to solve these dynamic routing problems, explore its operation with several general problem classes, and look at the advantages it provides in sample UAV routing problems. The end routine provides routing information for a UAV virtual battlespace simulation and allows dynamic routing of operational missions.

Keywords: Air Force Research, Operations Research, Optimization, Combinatorial Analysis, Algorithms, Remotely Piloted Vehicles, Surveillance Drones, Crosswinds, Ground Speed (Tabu Search, Vehicle Routing Problem, Java, Heuristics, Traveling Salesman Problem).
Chapter 1

The research documented in this thesis was sponsored by the Unmanned Aerial Vehicle (UAV) Battlelab. The UAV Battlelab is one of six battlelabs tasked with rapidly advancing warfighting concepts that enhance Air Force core competencies (Bailey 1998). The UAV Battlelab seeks innovative ideas and concepts with a military utility and attempts to quickly study and demonstrate those with promise. This evaluation is accomplished within the constraints of a limited budget and uses existing technologies whenever possible.

Improvement ideas, or initiatives, fall into one of two classes, Mitchell Class and Kenney Class. Mitchell Class initiatives are revolutionary in their impact and are typically costly. Kenney Class initiatives are usually straightforward and less costly (Bailey 1998). The research detailed in this paper falls under the auspices of a Kenney Class initiative, specifically one investigating potential UAV use for an active Suppression of Enemy Air Defense (SEAD) mission. The use of modeling and simulation resources has been identified as a crucial part of evaluating this new mission concept.
To aid in the modeling and simulation of the SEAD mission, the reactive tabu search heuristic route solver interfaces with a virtual UAV battlespace simulation created by Walston (1999). Her virtual battlespace simulates the US Air Force RQ-1A Predator in a SEAD mission and allows evaluation of tactics and aircraft design parameters.

A heuristic with the power to solve routing problems for simulation has the benefit of being able to solve *real-world* routing problems on-the-fly. Operational interface to the solver is available through a graphical user interface (GUI) (Flood 1999). The GUI provides air vehicle operators (AVOs), who are US Air Force pilots, the ability to dynamically update vehicle routes to reflect real-time changes in operational missions. This routing tool is offered as a proof of concept for possible implementation in their Ground Control Station (GCS) software.

Our primary objective was to give the Battlelab a robust and powerful routing black-box capable of dynamically supporting real-time operations and interfacing with the battlespace simulation. Additionally, we sought to provide a generic, portable, and expandable heuristic capable of solving this extremely difficult type of problem class, as well as others that are even *more* difficult. We have succeeded with a Reactive Tabu Search Heuristic implemented in the Java programming language.

The reactive tabu search is crucial to the success of both of these tools—tools that will provide the UAV community with a new modeling and simulation capability *and* provide the pilots with a new routing solver to aid flight mission planning. Both of these, in the long run, save money and enhance combat effectiveness.

This thesis research involved creating a Java implementation of a reactive tabu search (Battiti and Tecchiolli 1994, Carlton 1995, Ryan 1998) capable of solving
single/multiple traveling salesman problems with and without time windows (TSP, MTSP, TSPTW, MTSPTW), as well as capacitated vehicle routing problems with and without time windows (VRP, VRPTW). The RTS solver adds capacity constraints to Ryan's work, and extends Carlton's work with a reactive penalty scheme. Work on this RTS was done jointly with Ryer (1999) and results are representative of our combined efforts. Consequently, some items appear concurrently in both papers.

This implementation supports both classical problems and formats and UAV problems and formats. Changes required for the UAV problem reflect unique aspects of the operational mission, and include items such as a reformulated objective function, alternate coordinate and numerical formatting, and random customer service times. The introduction of altitude-based wind tiers, when selecting UAV routes, capitalizes on the altitude-dependent, highly asymmetric nature of travel times in an airborne environment.

The Java implementation is an object-oriented structure that is both machine portable and readily modifiable to support new problem instances. The internal data structure and methodology work with the GUI to support operational requirements such as route locking and dynamic rescheduling in support of priority targets.

This thesis is organized such that Chapter 2 is a stand-alone article on the research suitable for submission to an academic journal. Chapter 3 provides ideas that represent natural and worthwhile extensions to the work accomplished. The appendices provide in-depth information on various aspects of the research and other supplementary material. Appendix A provides detailed formulation for several routing class problems. Appendix B shows the performance of our reactive tabu search compared to other lesser heuristics on a sample traveling salesman problem. Appendix C gives documentation
that accompanies our Java code in the *JavaDoc* format. Appendix D provides additional references that were used in the course of the research, but are not cited or quoted in the main document.
Chapter 2

2.1 Introduction

Unmanned Aerial Vehicle (UAV) routing is a complex problem, and earlier work on the subject examined essentially predefined static scenarios. A tabu search coupled with a Monte Carlo Simulation was used to find the minimum number of vehicles required based on stochastic survival probabilities (Sisson 1997). Stochastic simulations involved selecting the best predefined route based on expected values of service, wind, and survival variables (Ryan 1998). This produced a robust tour which could then be used to mission plan a given set of targets with unknown threat and wind conditions at the time of mission execution. This approach is wholly appropriate for an autonomous UAV which is preprogrammed to execute a planned mission. While this gives a good starting point for a route schedule, it does not incorporate the latest information—information that can rapidly change.

The continuously evolving mission is a primary concern, especially to the operators of a long-duration, unmanned aerial vehicle such as the US Air Force’s RQ-1A Predator. An ability to dynamically adapt to the latest target update is fundamental to successful military operations. Therefore, we seek to take maximum advantage of current information (winds, target locations, threats, priorities) to dynamically generate and update routes for real-time use. This requires a method fast enough to be operationally effective, robust enough to handle a wide scope of problems, and reliable enough to provide optimal (or near optimal) solutions.
Most routing problems are NP-hard combinatorial problems for which no polynomially bounded algorithm has been found (Bodin et al. 1983). Convergent algorithms can rarely solve large problems consisting of more than 50 customers and often require relatively few side constraints (Gendreau et al. 1997). Unfortunately, real-world problems, such as UAV routing, possess many side constraints such as route and vehicle capacities, route length restrictions, and time windows in a sizeable network. Additionally, this network may be comprised of multiple depots and heterogeneous vehicles. Finding optimal solutions to these types of problems by using techniques such as branch and bound or dynamic programming is currently not practical.

Several heuristic approaches have been used in an attempt to overcome these problems. Greedy algorithms, which prove to be very useful in simpler problems, fail to achieve the desired results with respect to solution quality. Simulated annealing (SA) displays large variances in computational time and solution quality due to the random nature of its search strategy (Osman 1993). Genetic algorithms (GAs), which are designed to solve numerical optimization problems rather than combinatorial optimization problems, are difficult to apply to vehicle routing problems (VRPs) that require capacity, distance, and time window constraints (Gendreau et al. 1997). Fortunately, tabu search (TS) (Glover 1989) provides excellent results on these types of problems. The tabu search heuristic uses adaptive memory structures as it searches the solution space. Moves from one solution to another are made in a forced and orderly manner, and this forced move methodology allows the tabu search to escape the local extreme points. At each iteration, the tabu search will select a solution from the neighborhood provided the new candidate solution is not on the tabu list. The tabu list is
a data structure which keeps track of past solutions visited so that new solutions must be examined. Since the search must pick a new solution at each iteration, the items on the tabu list will be tabu, or off-limits, and the heuristic will pick the best non-tabu move, which may actually be a worse solution. This seems somewhat counter-intuitive, but the search will continue on to find unexplored areas which potentially may yield better overall results. A special instance called aspiration allows the tabu status of a move to be overruled if certain conditions are met. The tabu status will be overridden and the solution accepted if it is deemed good enough based on certain attractiveness thresholds. The length of time a solution stays on the tabu list is determined by the tabu list length. Based on the length of the tabu list, the behavior of the search can be significantly altered. If the list is shortened, intensification occurs and the local area will be searched more thoroughly as the search gravitates towards the local optimum. If the list is lengthened, diversification occurs and the search will be forced leave its current area to explore new areas further away in the solution space (Glover 1997).

The literature shows TS is a robust approach to solving many variations of the VRP and dominates current studies of routing problems (Garcia et al. 1994, Osman 1993, Rochat and Semet 1994, Carlton 1995, Xu and Kelly 1996, Chiang and Russell 1997, Gendreau et al. 1997, Barbarosoglu and Ozgur 1999). Even certain vehicle routing methods, such as the sweep method and petal heuristic, are not as powerful as tabu search algorithms (Renaud et al. 1996b).

This project explores the application of the reactive tabu search (RTS) metaheuristic to routing problems, specifically the vehicle routing problem with time windows (VRPTW). Our RTS follows the basic TS scheme, but differs in that it actively
adjusts the tabu length based on the quality of the search, as determined by the number of iterations before a solution is revisited. In execution this project implements the object-oriented (OO) Java programming language for two reasons. First, the OO design of software allows us to reuse and modify existing code and libraries which reduces the development time of new software routines to extend problems (Eckel 1998). Second, Java programs offer a cross-platform compatibility which enhances portability. Our Java heuristic implementation follows, improves, and extends a MODSIM implementation (Ryan 1998) based on an RTS developed by Battiti and Tecchiolli (1994) and implemented by Carlton (1995).

In this paper, first we examine a reactive tabu search heuristic suitable for solving traveling salesman and vehicle routing problems and provide our results from a Java implementation of this solver. We look at enhancements to the RTS, the verification and validation results, and explore how this tabu search successfully solves tough problems. We review past work and general formulation of the UAV routing problem. We look at our modifications to previous efforts and show how the RTS enables us to solve this problem in particular. Finally, we suggest areas for future exploration.

2.2 Reactive Tabu Search for the Vehicle Routing Problem with Time Windows

The vehicle routing problem with time windows (VRPTW) is defined as follows: Let \( G = (V, A) \) be a graph where \( V = \{v_0, v_1, \ldots, v_n\} \) is the vertex set and

\[
A = \{ (v_i, v_j) : v_i, v_j \in V, i \neq j \}
\]

is the arc set. The depot vertex \( v_0 \), has \( m \) identical vehicles, each with a maximum load capacity \( Q \) and a maximum route duration \( D \). The
remaining vertices $v_i \in V$ represent customers to be serviced, each with a non-negative demand $q_i$, a service time $s_i$, and a service time window comprised of a no-earlier-than time $e_i$ and a no-later-than time $\ell_i$. The no-earlier-than time window constraint is considered soft, i.e., an arrival time $a_i$ before the early time results in a wait time $w_i$ until $e_i$ to commence service. Each edge $(v_i, v_j)$ has an associated non-negative cost $c_{ij}$, interpreted as travel time $t_{ij}$ between locations $i$ and $j$.

The objective of the vehicle routing problem with time windows (VRPTW) is to determine a set of $m$ vehicle routes starting and ending at the depot, such that each customer is visited exactly once within its time window, the total demand of any vehicle route does not exceed $Q$, the duration of any vehicle route does not exceed $D$, and the total cost of all routes is minimized. When only one vehicle is available and $Q$, $D$, $e_i$, and $\ell_i$ are non-binding constraints, the problem reduces to a traveling salesman problem (Renaud et al. 1996a).

A tour is defined by the order in which the $n$ customers are served by the $m$ vehicles. In our heuristic, we represent the problem as an ordered list of the sequence of customers and vehicles, or disjunctive graph, as shown in Figure 1.

![Disjunctive Graph Notation](image)

Figure 1. Disjunctive Graph Notation
The first and last positions ($0$ and $n + m$) in this sequence represent the initial depot/vehicle and an additional terminal depot required to close the graph. These two nodes are fixed and will not move during the search. Initially, the customers occupy positions between $1$ and $n$ and the additional vehicles occupy the remaining positions between $n + 1$ and $n + m - 1$ as shown in Figure 2. During the search, customers and vehicles will be interspersed, and unused vehicles will occupy positions between the last serviced customer and the final depot.

\[ T(d) = \begin{array}{cccccccc}
0 & 1 & 2 & \ldots & n & n+1 & \ldots & n+m-1 & n+m \\
\text{Vehicle Nodes} & \text{Customer Nodes} \\
\end{array} \]

Figure 2. Initial Tour Sequence

### 2.2.1 Objective Function.

For the generic VRPTW, we seek to minimize travel costs $c_{ij}$ along the selected arcs identified by $x_{ij} = 1$. This is given by

\[
\text{minimize } Z(t) = \sum_j \sum_i c_{ij} x_{ij} \tag{1}
\]

Where $X = (x_{ij}) \in S$, $x_{ij} \in \{0,1\}$ $\forall i, j$.

Full enumeration of all constraints is available in Appendix A.
2.2.2 Penalized Objective Function.

A major advantage of our method is that it effectively explores the solution space by considering both feasible and infeasible solutions. First, instead of being restricted only to feasible regions, our RTS can traverse regions of infeasibility to include starting with an infeasible initial solution. Second, the infeasible solutions generated may be used in real world applications with flexible constraints. For instance, an infeasible solution that produces superb overall results may become feasible with the relaxation of a constraint controlled by the decision-maker. Such a case occurred with a delivery problem solved by Rochat and Semet (1994). Since very few real-world constraints are absolutely hard, these infeasible solutions may represent some difficult route selection choices that managers may face when trying to balance competing criteria.

A solution is infeasible if it violates a time window, load capacity, or duration constraint. Constraint violations include missed time windows $TW$ and excess vehicle load capacity $LD$ defined as

$$TW = \sum_i \left[ \max(0, a_i - \ell_i) \right] + \sum_i \left[ \max(0, a_i - D_i) \right]$$

and

$$LD = \sum_i \left[ \max(0, q_i - Q_i) \right]$$

respectively. Each constraint violation is scaled by a corresponding penalty factor, $\rho_{tw}$ and $\rho_{ld}$, giving the penalized objective function as

$$\text{minimize} \quad Z(t) = Z_f(t) + \rho_{ld} LD + \rho_{tw} TW \quad (2)$$
where $Z_f(t)$ is the original objective function given by (1). If the solution is feasible, then $Z_f(t)$ and $Z(t)$ are equivalent. Otherwise, $Z(t)$ will include non-zero penalty terms.

2.2.3 Adjusting Reactive Penalty Coefficients.

The penalty factors should be large enough to separate the infeasible and feasible regions of the solution space so that infeasible solutions do not dominate feasible solutions. The penalty factors should also be small enough to allow consideration of infeasible solutions. Appropriate penalty values can be very difficult to calculate (Petridis et al. 1998), so our implementation allows for self-adjusting penalty values in addition to constant user-set penalty values.

When self-adjusting, the value of the penalty coefficients $\rho_{LD}$ and $\rho_{TW}$ are independently adjusted every five iterations as proposed by Gendreau et al. (1996) using the relationship

$$\rho_{TW} = \rho_{TW} \cdot \left(\frac{t_{TW}}{5}\right)^{t_{TW} - 1},$$

$$\rho_{LD} = \rho_{LD} \cdot \left(\frac{t_{LD}}{5}\right)^{t_{LD} - 1},$$

where $t_{TW}$ is the number of time window infeasible solutions among the last ten solutions and $t_{LD}$ is the number of capacity infeasible solutions among the last ten solutions. If all ten previous solutions are feasible, the current $\rho$ is multiplied by $\frac{1}{2}$. If all ten previous solutions are infeasible, the current $\rho$ is multiplied by 2. Intermediate numbers of infeasible solutions yield multiplicative factors between $\frac{1}{2}$ and 2. The penalty values are arbitrarily limited to the closed interval $[0.1, 10^{100}]$, a range easily represented by Java.
This prevents the penalties from being rounded by Java to unadjustable zero or infinity values. In the reactive penalty scheme, we arbitrarily set both penalty values initially to 1000.

The reactive penalties provide a measure of trajectory control into and out of feasible regions based on the collective feasibility of the previous solutions. When many successive solutions are feasible, the lowered penalties do not strongly discourage movement to an infeasible solution. Successive infeasible solutions drive the penalties higher, putting increasingly greater emphasis on finding a feasible solution.

2.2.4 Initial Solution.

An initial solution, which may or may not be feasible, is arbitrarily constructed. We employ three options for arranging this initial solution—the first is a listed ordering, the second is based on the time window midpoint, and the third is based on a randomized ordering. All three methods arbitrarily construct a solution by assigning all customers to one vehicle.

The list ordered tour method (LOT) simply assigns customers to the vehicle in the order that they are listed in the data set. The ordered starting tour (OST) method generates a starting solution by sorting the customers based upon increasing time window midpoint values while enforcing the time window feasibility conditions. The time window midpoint for the customer $i$ is defined as halfway between $e_i$ and $\ell_i$.

The random starting tour (RST) method randomly reorders the sequential starting list of customers to provide a different starting point. Since the tabu search is a neighborhood search, the initial starting solution will influence the progression of the
search. Our experimentation suggests that the reactive tabu search is robust and relatively insensitive to the initial tour.

2.2.5 *Neighborhood Structure.*

Our solution neighborhood is the set of tours immediately reachable from the current solution with a single 3-opt move. The 3-opt move removes three edges and replaces them with three new edges in a way that moves one vertex to another location in the tour sequence. From the disjunctive graph formulation, the solution neighborhood is examined with incremental *swap* moves and updated with an *insertion* move. A swap move exchanges the position of two adjacent nodes with a 3-opt move as shown in Figure 3. An insertion move relocates a specific customer at location *i* forwards or backwards in the tour by a number of steps called the insertion depth *d*. In our implementation, an insertion is executed as a series of sequential swap moves.

![Initial Sequence before Swap](image1)

![Updated Sequence after Swap](image2)

**Figure 3.** Adjacent 3-Opt Swap Move
This problem types yields a staggering \((n + m - 1)!\) possible solution permutations—a relatively simple 25 customer, 5 vehicle problem has \(8.842 \times 10^{30}\) possible solutions. To reduce the neighborhood size, moves which result in a redundant tour are prohibited. Additionally, strong time window feasibility is enforced (Carlton 1995).

Strong time window infeasible states occur between nodes \(i\) and \(j\) whenever a vehicle leaving node \(i\) at departure time \(d_i\) can never arrive at node \(j\) within the required time window. Specifically, node \(j\) is strong time window infeasible with respect to node \(i\) if \(d_i + t_{ij} > \ell_j\ \forall\ d_i = a_i + s_i, \ a_i \in [e_i, \ell_i]\). Weak time window infeasible states occur when only some departure times preclude a timely arrival at the following node, i.e., \(d_i + c_{ij} \leq \ell_j\ \forall\ d_i = a_i + s_i, \ a_i \leq t, t \in [e_i, \ell_i]\). Unlike strong time window infeasible tours, weak time window infeasible tours are evaluated in the search since insertion moves can ultimately reduce the amount of infeasibility in the overall tour (Carlton 1995). Past vehicle routing problem research indicates that feasible solutions may be isolated or disjoint from each other in the solution space, so in order to effectively search the solution space, the method must investigate and perhaps accept infeasible solutions. This search of the infeasible region is facilitated by our use of penalty factors.

2.2.6 Tabu Moves.

Tabu search uses a memory structure to determine if a particular tour has already been visited by examining its attributes. The examination must efficiently and reliably store and identify solution attributes previously altered during the search. We employ an
(n + 1) \times (n + 1) \text{ dimension Tabulist matrix with rows corresponding to customer identification numbers and columns corresponding to the index, or position, of the customer in the solution tour. The data elements in this array store the iteration number } k \text{ for the move that placed the customer into this position plus the tabu length } \theta. \text{ This value will be compared to the current iteration to determine if a move of this attribute is tabu.}

2.2.7 Adjusting Tabu Length.

To maintain search quality, we reactively adjust the tabu length based on the number of iterations occurring between cycles. Cycles occur when the search revisits a solution; a high quality search should infrequently revisit past solutions. Given the combinatorial nature of the problem, it is possible to select a seemingly different tour that is actually a redundant tour—one that appears new, but in fact is a revisit of a previous equivalent tour. Figure 4 illustrates two different tours which are actually redundant tours.

Redundant tours are identified with a two-attribute hashing scheme. The first hashing attribute, the hashing function } f(t), \text{ is assigned the objective function value } Z(t). \text{ Woodruff and Zemel (1993) propose a method that we use to compute the second hashing attribute, the tour hashing value } thv(t). \text{ We take the tour vector and calculate an integer based on random integer values, } \Psi(\tau_i), \text{ where } \tau_i \text{ is the index of the customer assigned to tour position } i, \text{ such that}

\[ thv(T) = \sum_{i=0}^{n} \Psi(\tau_i) \cdot \Psi'(\tau_{i+1}) \]
This tour hashing value attempts to minimize the occurrence of a collision, or the incorrect identification of two tours as being identical or redundant when they are actually distinct.

\[ T(1) = 0 \rightarrow 1 \rightarrow 2 \rightarrow 6 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 5 \rightarrow 8 \]

\[ T(2) = 0 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 5 \rightarrow 7 \rightarrow 1 \rightarrow 2 \rightarrow 8 \]

Figure 4. Redundant Tours

We also use other attributes to identify a solution; these are tour cost, travel time, time window penalty, and total penalty. These integer values are concatenated into a uniquely identifiable Java string object and stored with Java Hashtable class functions. This unique string value allows us to efficiently identify past solutions, as well as access the hash record containing solution attributes stored in their original form.

When the search revisits a solution within the designated number of iterations, or cycle length, the tabu length is increased by a scaling factor. This tabu length increase diversifies the search. If the search is not revisiting solutions, tabu length is decreased by a scaling factor. When a solution is revisited within the maximum cycle length, the algorithm calculates a moving average of cycle lengths, or the average number of iterations between a revisit. If the tabu length has remained unchanged for a number of iterations greater than or equal to this moving average, then the current tabu length is decreased by the scaling factor, thus intensifying the search. We set the initial tabu length value \( \theta \) to the smaller of either 30 or \( m + n - 1 \).
2.2.8 Aspiration and Escape Functions.

Aspiration allows for overriding the tabu status of a move if the proposed tour solution is better than any previous solution. If all moves are tabu and no proposed solution meets aspiration criteria, the search escapes to the neighbor tour with smallest move value. This escape move is accomplished regardless of tabu status and results in a tabu length decrease.

2.2.9 Move Evaluation and Selection.

The RTS systematically explores the solution space using a series of swap moves and chooses the allowable adjacent solution with the smallest move value. The move value is the difference between the incumbent’s objective function value and the candidate’s objective function value given as the cost/travel savings resulting directly from the 3-opt move and the resultant changes occurring in the rest of the tour.

2.2.10 Heuristic Description.

Crucial to the success of the solver is the time matrix which contains the travel times $t_{ij}$ between every node combination $i, j$. The time matrix is built in a three-step process. First, cartesian distances between locations are computed. Second, these distances are converted to times based on problem parameters. Third, the service time at node $i$ is added to the time. As such, $t_{ij}$ values then represent the amount of time between arrival at node $i$ and the subsequent arrival at node $j$. We use these values as our costs, i.e., $c_{ij} = t_{ij}$. Actual en route travel time can be calculated by subtracting service time $s_i$ from $t_{ij}$.
The reactive tabu search executes the following steps.

*Step 1* (Initialization) Initialize data structures, vectors, and parameters.

*Step 2* (Problem Input) Read data and assign node information. Calculate appropriate time matrix.

*Step 3* (Route Initialization) Construct initial tour, calculate initial tour schedule, and compute associated tour cost and hashing value. Store values. Assign initial tour as incumbent tour.

*Step 4* (Cycle Check) Check hashing structure for the incumbent tour. If found, update the iteration when found, increase the tabu length if applicable. If not found, add to the hashing structure, decrease the tabu length, if applicable. Increment current iteration number.

*Step 5* (Check Later Insertions) Accomplish swap moves to evaluate all forwards insertions. Store position \( i \) and depth \( d \) of best move value, aspiration, and escape information.

*Step 6* (Check Earlier Insertions) Accomplish swap move to evaluate all backwards insertions. Store \( i, d \) of best move value, aspiration, and escape information.

*Step 7* (Execute Move) Move to a non-tabu neighbor according to appropriate decision criteria. If all moves are tabu, use the escape move and reduce the tabu length. Perform insertion, update schedule, assign neighbor tour as new incumbent tour, compute hashing value, and track best tour information. If current iteration number is less than the maximum iteration number, return to Step 4.

*Step 8* (Output results) Terminate heuristic search and output results.

2.2.11 Computational Complexity.

The neighborhood size considered at each step is \( O(nd) \), and the computation of the move value for each neighbor is \( O(n) \). If the depth of the insertion moves is restricted to 1, then the algorithm achieves a minimum computational complexity of \( O(n^2) \). The worst case complexity is \( O(n^2d) \) where \( d \) is the depth of the allowable insertion moves. When the insertion depth is expanded to \( n \) the computational complexity expands with it to a maximum \( O(n^3) \). However, empirical testing shows that considerably better times
than $O(n^3)$ can be achieved due to the strong time window infeasibility restriction discussed earlier (Carlton 1995).

2.3 Reactive Tabu Search for Dynamic Unmanned Aerial Vehicle Routing

The US Air Force uses the Predator UAV to perform a reconnaissance and surveillance mission. The Predator is remotely flown by Air Vehicle Operators, who are Air Force pilots, located in a Ground Control Station. Co-located Payload Specialists remotely control the electro-optical camera, infrared scanner, and synthetic aperture radar to observe targets of interest as specified by higher command elements. The imagery is returned real-time via satellite link to intelligence specialists and regional commanders (McKenna 1998). The Predator has been used successfully to monitor buildings, military forces, and battle activities in Bosnia pursuant to United Nations and NATO missions. The Predator's long airborne endurance of nearly 40 hours and its ground based control system (with ready access to computers) makes it an ideal candidate for efficient computerized routing strategies.

We seek to enhance the capabilities of existing mission software. Current software will automatically generate deterministic items such as terrain avoidance profiles, ground station to UAV line of site availability, route times between defined way points, fuel consumption, heading and turn information, etc., but it does not and will not optimize routes. This combinatorial problem is a task left to the operator. We provide our routing tool to fill the gap that exists in making complicated routing decisions.
Since this is a real-world operational problem, several real-world operational factors influence our implementation approach.

2.3.1 Operational Parameters.

Operational employment of the UAV drives several changes to how the problem data is specified and solved. These changes range from relatively superficial ones in how the coordinates and times are represented, to moderate changes in how the parameters are calculated, to significant changes in how the objective function is formulated to reflect the nature of the problem.

2.3.2 Geographic Coordinates.

Coordinates are expressed in a geocentric format instead of a Cartesian format. We calculate the distance and bearing between coordinate points as shown in AFR 51-40, Air Navigation (Departments of the Air Force and Navy 1983). Given the departure latitude \( L_1 \) and longitude \( \lambda_1 \) and the destination latitude \( L_2 \) and longitude \( \lambda_2 \), the great circle distance \( d \) in nautical miles between the two coordinate points can be found using the following formulation

\[
d = 60 \cdot \cos^{-1} \left[ \sin L_1 \cdot \sin L_2 + \cos L_1 \cdot \cos L_2 \cdot \cos(\lambda_2 - \lambda_1) \right] .
\]

Using this distance, an intermediate heading angle \( H \) in degrees is determined as

\[
H = \cos^{-1} \left[ \frac{\sin L_2 - \sin L_1 \cdot \cos \left( \frac{d}{60} \right)} {\sin \left( \frac{d}{60} \right) \cdot \cos L_1} \right].
\]

Based on the geometry of the coordinates, this intermediate heading angle is adjusted to
obtain the initial true heading $\Theta_j$, measured in degrees from true north, i.e.,

$$\Theta_j = \begin{cases} H, & \sin(\lambda_2 - \lambda_1) < 0 \\ 360^\circ - H, & \sin(\lambda_2 - \lambda_1) \geq 0 \end{cases}.$$

This distance and bearing geometry is shown in Figure 5.

![Figure 5. Distance and Bearing Geometry (Spherical Triangle)](image)

2.3.3 Wind Effects on Ground Speed.

When computing transit times between locations, we must account for the effect of winds aloft. Given a wind speed $WS$ from a bearing of $\Theta_{WS}$ measured in degrees from true north, one can calculate the effective ground speed $GS$ along the true course $\Theta_j$ from the first location to the second. The difference between $\Theta_j$ and $\Theta_{WS}$ is represented by $\delta$.

Figure 6 illustrates this geometry. When $|\delta| < 90$, $A$ is negative and subtracts from the airspeed as a headwind component. When $90 < |\delta| \leq 180$, $A$ is positive and adds to the airspeed as a tailwind component. The wind correction angle from true heading is denoted by $\gamma$. This adjusted heading corrects the flight path to compensate for wind drift. Groundspeed as influenced by wind aloft, is explicitly calculated as follows.

$$\delta = \Theta_j - \Theta_{WS}$$
\[ A = WS \cdot \cos(180 - \delta) \]
\[ C = WS \cdot \sin(180 - \delta) \]
\[ B = \sqrt{AS^2 - C^2} \]
\[ GS = A + B = WS \cdot \cos(180 - \delta) + \sqrt{AS^2 - WS^2 \cdot \sin^2(180 - \delta)} \]

The transit time between the points is then simply \( t = d_y / GS \).

![Diagram of Headwind and Tailwind Ground Speed Adjustment](image)

Figure 6. Headwind and Tailwind Ground Speed Adjustment

### 2.3.4 Numerical Formatting.

The latitude \( L \) and longitude \( \lambda \) information is measured in degrees where one degree is composed of sixty minutes and one minute is composed of sixty seconds. The values are often listed in a *degrees minutes seconds* format (DD MM SS.ss); we convert
latitudes and longitudes into a decimal degree format for computational ease using the formula

\[ D.d = DD + MM/60 + SS\cdot ss/3600 \]

Locations can also be listed in a degrees minutes decimal minutes format (DD MM.mm) where minutes are expressed as decimal values. Conversion to a decimal degree value is defined as

\[ D.d = DD + MM.mm/60 \]

Clock time is often expressed in a military-style hours minutes (HH MM) format; for computational ease, we express time in minutes \( t_{\text{minutes}} \) as

\[ t_{\text{minutes}} = 60 \cdot HH + MM \]

2.3.5 Objective Function Modifications.

The UAV operating environment also mandates changes to the objective function. The standard VRPTW objective function seeks to minimize travel costs as represented by the distance traveled. Early arrival to a customer is allowed, and the resulting waiting time is cost free in the objective function. This may be appropriate for a standard terrestrial application in which the costs are associated mainly with transiting between locations, but in UAV operations there are costs associated with keeping the aircraft airborne. Thus, UAV waiting times represent costs that must be considered in our efforts to minimize the objective function. We therefore modify the original and penalized objective functions (1) and (2) to include waiting time \( w_i \) at node \( i \) in addition to the

24
original transit times as

\[
\text{minimize } Z'(t) = \sum_j \sum_i (c_{ij} + w_i) x_{ij}.
\]

As before, the penalized objective function is gained by adding the scaled infeasibility values to yield

\[
\text{minimize } Z'(t) = Z'(t) + \rho_{LD} LD + \rho_{TW} TW.
\]

The search now attempts to minimize the total time aloft and proceeds as previously presented.

2.3.6 Dynamic Mission Requirements.

The nature of UAV employment presents unique situations that our routing tool must handle. As such, we show how our scenarios depart from traditional VRPs and explain how we successfully implement these requirements. Unique routing situations exist with regard to altitude-based wind tiers, random service times, emerging priority targets, and locked route sequences. These instances are explored in the following paragraphs.

2.3.7 Optimizing Use of Altitude-Based Wind Tiers.

Recall that in the general MTSPTW problem, travel times between fixed locations are known, with fixed and symmetric costs (i.e., \( c_u = c_{\bar{u}} \)). This symmetry does not hold in the UAV operating environment where winds affect travel times and can vary both in direction and velocity as a function of altitude (depicted below in Figure 7). We incorporate this wind information to select the minimum travel time between nodes.
Specifically in any altitude band \( k \), travel time is a function of UAV altitude \( h_{ijk} \) and airspeed \( AS_k \); wind speed \( WS_k \) and direction \( \Theta_{WSk} \); and the distance \( d_y \) and bearing \( \Theta_y \) between locations. We make the simplifying assumption that wind direction and speed is constant throughout an entire altitude zone. This is reasonable since values at any point in the region are interpolated predictions based on measurements of actual conditions at discrete weather station locations (Parsons 1999).

Using our previous equations, we calculate times between all locations based on the adjusted ground speed for each altitude band. This forms tiers of asymmetric wind-influenced travel time matrices from which we select the smallest travel time from \( i \) to \( j \) as

\[
t_{ij} = \min_{\forall k} \left[ \frac{d_y}{GS_y(h_{ijk})} \right]
\]

where \( GS_y(h_{ijk}) \) the ground speed as a function of traveling in altitude band \( k \). The corresponding altitude is assigned as our flight altitude for that leg. Since this wind optimization process is accomplished prior to beginning the tabu search, the heuristic will
accept an arbitrary number of altitude bands with no appreciable effect on computational time or efficiency.

2.3.8 Random Service Times.

In the general TSPTW problem, customer service times are known constants. In the UAV problem context, the target service times are random variables. The service time represents the amount of time the aircraft spends circumnavigating the target point to gather imagery from multiple viewpoints, and, due to the unknown nature of the target, military necessity may dictate a longer observation than initially planned. The actual target $i$ service time $S_i$ falls between the minimum service time $s_{\text{min}}(i)$ and the maximum service time $s_{\text{max}}(i)$ inclusive. Service time will be the minimum service time with 0.7 probability; when the time is above the minimum, it is modeled as uniformly distributed between the minimum and maximum. The service time is given by

$$ S_i = \begin{cases} 
  s_{\text{min}}(i) & \text{with 0.7 probability} \\
  \text{Uniform}(s_{\text{min}}(i), s_{\text{max}}(i)) & \text{with 0.3 probability} 
\end{cases}.$$ 

A known service time is simply specified by setting $S_i = s_{\text{min}}(i) = s_{\text{max}}(i)$.

2.3.9 Emerging Targets.

Another aspect of UAV operations is the pop-up priority target. This occurs when the UAV is retasked in flight to observe a target of utmost military urgency. Depending on the new target location, this immediate divert may render the remainder of the route
obsolete. Rather than proceed with a potentially sub-optimal route, our solver offers the ability to route-from-here.

Given that the UAV will proceed to the ad hoc target, this location becomes a new starting point and the remaining targets are processed in a route that returns the UAV to the depot. This route-from-here capability is achieved with smart processing of the time matrix.

2.3.10 Locked and Forbidden Routes.

At times, UAV operations require a locked route, in which one or more targets must be visited in a specific order. This may occur with a directed route or with certain observational requirements such as a consecutive imaging pass for a synthetic aperture radar image. The GUI allows these route points to be locked together and treated as an aggregated node with a beginning location corresponding to the first point and an ending location corresponding to the last point. The aggregated node is assigned a composite service time that accounts for intra-node service, wait, and travel times.

The opposite of a locked route is a forbidden route which may be a result of a no-fly zone or threat region. The forbidden area is then monitored for flight paths which pass through it; if a path intersects a forbidden area, it is modeled as a longer route that skirts the edge of the region as shown in Figure 8.
2.4 Computational Results

2.4.1 General Results.

Our initial testing and validation used the Solomon VRPTW problem test sets—25, 50, and 100 customer scenarios with random, clustered and random clustered distribution patterns. Our computational results are compared in Tables 1-6 (Ryer 1999) to known optimal answers obtained by Desrochers, Desrosiers, and Solomon (1992). Dashed regions of the chart indicate problems that could not be optimally solved by Desrochers et al. All problems were solved in reasonable computation times by our RTS algorithm (2500 iterations with user specified penalties) with an overall solution quality within 1% of optimal values. Solving the harder VRPTW class problems did not require an increase in computation times over the mTSPTW class problems.

The objective function value used in these initial tests includes travel time, missed time window penalties, and load overage penalties. With a relatively small amount of coding, the objective function can be expanded to include additional penalties, changed to represent several different weighted objective functions, or combined in a hierarchical objective function. Results are presented in Tables 1 through 6.
<table>
<thead>
<tr>
<th>Problem Set</th>
<th>O’Rourke &amp; Ryer</th>
<th>Optimal</th>
<th>Difference</th>
<th>Start Method*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( Z(t) ) Used</td>
<td>Iter^2</td>
<td>Time^3</td>
<td>( Z(t) ) Used</td>
</tr>
<tr>
<td>R101</td>
<td>867.1</td>
<td>8</td>
<td>317</td>
<td>3</td>
</tr>
<tr>
<td>R102</td>
<td>797.1</td>
<td>7</td>
<td>35</td>
<td>1</td>
</tr>
<tr>
<td>R103</td>
<td>704.6</td>
<td>5</td>
<td>132</td>
<td>1</td>
</tr>
<tr>
<td>R104</td>
<td>666.9</td>
<td>4</td>
<td>86</td>
<td>1</td>
</tr>
<tr>
<td>R105</td>
<td>780.3</td>
<td>6</td>
<td>95</td>
<td>1</td>
</tr>
<tr>
<td>R106</td>
<td>715.4</td>
<td>5</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>R107</td>
<td>674.3</td>
<td>4</td>
<td>2080</td>
<td>23</td>
</tr>
<tr>
<td>R108</td>
<td>647.3</td>
<td>4</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>R109</td>
<td>691.3</td>
<td>5</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>R110</td>
<td>694.1</td>
<td>5</td>
<td>91</td>
<td>2</td>
</tr>
<tr>
<td>R111</td>
<td>678.8</td>
<td>4</td>
<td>178</td>
<td>2</td>
</tr>
<tr>
<td>R112</td>
<td>643.0</td>
<td>4</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>C101</td>
<td>2441.3</td>
<td>3</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>C102</td>
<td>2440.3</td>
<td>3</td>
<td>379</td>
<td>4</td>
</tr>
<tr>
<td>C103</td>
<td>2440.3</td>
<td>3</td>
<td>72</td>
<td>1</td>
</tr>
<tr>
<td>C104</td>
<td>2436.9</td>
<td>3</td>
<td>797</td>
<td>8</td>
</tr>
<tr>
<td>C105</td>
<td>2441.3</td>
<td>3</td>
<td>209</td>
<td>2</td>
</tr>
<tr>
<td>C106</td>
<td>2441.3</td>
<td>3</td>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>C107</td>
<td>2441.3</td>
<td>3</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>C108</td>
<td>2441.3</td>
<td>3</td>
<td>1421</td>
<td>15</td>
</tr>
<tr>
<td>C109</td>
<td>2441.3</td>
<td>3</td>
<td>148</td>
<td>1</td>
</tr>
<tr>
<td>RC101</td>
<td>711.1</td>
<td>4</td>
<td>214</td>
<td>3</td>
</tr>
<tr>
<td>RC102</td>
<td>661.7</td>
<td>3</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>RC103</td>
<td>582.8</td>
<td>3</td>
<td>2193</td>
<td>24</td>
</tr>
<tr>
<td>RC104</td>
<td>556.6</td>
<td>3</td>
<td>604</td>
<td>6</td>
</tr>
<tr>
<td>RC105</td>
<td>661.2</td>
<td>4</td>
<td>79</td>
<td>1</td>
</tr>
<tr>
<td>RC106</td>
<td>595.5</td>
<td>3</td>
<td>60</td>
<td>1</td>
</tr>
<tr>
<td>RC107</td>
<td>548.3</td>
<td>3</td>
<td>69</td>
<td>1</td>
</tr>
<tr>
<td>RC108</td>
<td>544.5</td>
<td>3</td>
<td>2203</td>
<td>23</td>
</tr>
</tbody>
</table>

Average 1218.19 3.93 402.7 4.38 1184.8 3.90 148.6 0.69 0.11% —

(Ryer 1999)

1 Maximum number of vehicles: \( m = 10 \). Time window penalty: \( \rho_{w} = 1.0 \).

2 Maximum iterations: \( k = 2500 \).

3 Seconds on a Pentium III 400 MHz system. Total runtime ~ 28 seconds each.

4 Seconds on a Sun Sparc 1 workstation.

5 OST is ordered starting tour. RST is random starting tour seeded with the value given. LOT is listed ordering.
Table 2. Solomon mTSPTW Computational Results (50 Customers)

<table>
<thead>
<tr>
<th>Problem Set</th>
<th>O’Rourke &amp; Ryer</th>
<th>Optimal</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Z(t) Used</td>
<td>Iter $^2$</td>
<td>Time $^3$</td>
</tr>
<tr>
<td>R101</td>
<td>1543.8</td>
<td>12</td>
<td>239</td>
</tr>
<tr>
<td>R102</td>
<td>1409.0</td>
<td>11</td>
<td>1939</td>
</tr>
<tr>
<td>R103</td>
<td>1282.7</td>
<td>9</td>
<td>871</td>
</tr>
<tr>
<td>R104</td>
<td>1131.9</td>
<td>6</td>
<td>734</td>
</tr>
<tr>
<td>R105</td>
<td>1401.6</td>
<td>9</td>
<td>402</td>
</tr>
<tr>
<td>R106</td>
<td>1293.0</td>
<td>8</td>
<td>2294</td>
</tr>
<tr>
<td>R107</td>
<td>1211.1</td>
<td>7</td>
<td>1786</td>
</tr>
<tr>
<td>R108</td>
<td>1117.7</td>
<td>6</td>
<td>1698</td>
</tr>
<tr>
<td>R109</td>
<td>1286.7</td>
<td>8</td>
<td>1452</td>
</tr>
<tr>
<td>R110</td>
<td>1207.8</td>
<td>7</td>
<td>1853</td>
</tr>
<tr>
<td>R111</td>
<td>1216.6</td>
<td>7</td>
<td>1775</td>
</tr>
<tr>
<td>R112</td>
<td>1140.5</td>
<td>6</td>
<td>1784</td>
</tr>
</tbody>
</table>

| C101        | 4862.4 | 5 | 119 | 4 | 4862.4 | 5 | 67.1 | 0.0 | 0.00% | LOT |
| C102        | 4861.4 | 5 | 607 | 19 | 4861.4 | 5 | 330.2 | 0.0 | 0.00% | LOT |
| C103        | 4855.8 | 5 | 1699 | 57 | — | — | — | — | — | OST |
| C104        | 4884.1 | 5 | 1253 | 43 | — | — | — | — | — | LOT |
| C105        | 4861.2 | 5 | 232 | 7 | — | — | — | — | — | OST |
| C106        | 4862.4 | 5 | 308 | 9 | 4862.4 | 5 | 91.3 | 0.0 | 0.00% | LOT |
| C107        | 4861.2 | 5 | 382 | 12 | — | — | — | — | — | LOT |
| C108        | 4861.2 | 5 | 92 | 3 | — | — | — | — | — | LOT |
| C109        | 4860.9 | 5 | 301 | 9 | — | — | — | — | — | OST |

| RC101       | 1444.0 | 8 | 1252 | 38 | — | — | — | — | — | RST 1 |
| RC102       | 1325.1 | 7 | 754 | 23 | — | — | — | — | — | RST 1 |
| RC103       | 1216.2 | 6 | 1589 | 54 | — | — | — | — | — | RST 0 |
| RC104       | 1046.5 | 5 | 860 | 31 | — | — | — | — | — | RST 2 |
| RC105       | 1355.3 | 8 | 248 | 8 | — | — | — | — | — | OST |
| RC106       | 1223.2 | 6 | 1921 | 61 | — | — | — | — | — | RST 2 |
| RC107       | 1146.0 | 6 | 189 | 7 | — | — | — | — | — | LOT |
| RC108       | 1098.1 | 6 | 1821 | 65 | — | — | — | — | — | OST |

Average: 2374.7 | 6.66 | 1050 | 39.6 | — | — | — | — | — | — |

(Ryer 1999)

1 Maximum number of vehicles: R sets $m = 15$; C sets $m = 6$; RC sets $m = 8$. Time window penalty: $\rho_{tw} = 3.0$.
2 Maximum iterations: $k = 2500$.
3 Seconds on a Pentium II 400 MHz system. Total runtime ~ 100 seconds each.
4 Seconds on a Sun Sparc 1 workstation.
5 OST is ordered starting tour. RST is random starting tour seeded with the value given. LOT is listed ordering.
Table 3. Solomon mTSPTW Computational Results (100 Customers)

<table>
<thead>
<tr>
<th>Problem Set</th>
<th>O'Reurke &amp; Ryer</th>
<th>Optimal</th>
<th>Difference</th>
<th>Start Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Z(t) Used Iter$^2$ Time$^3$</td>
<td>Z(t) Used Time$^4$</td>
<td>$\Delta$ $A%$</td>
<td></td>
</tr>
<tr>
<td>R101</td>
<td>2689.6 20 2167 371</td>
<td>2607.7 18 1064.2</td>
<td>81.9 3.14%</td>
<td>RST 0</td>
</tr>
<tr>
<td>R102</td>
<td>2522.9 18 1783 322</td>
<td>2434.0 17 756.9</td>
<td>88.9 3.65%</td>
<td>RST 0</td>
</tr>
<tr>
<td>R103</td>
<td>2266.8 15 1797 351</td>
<td>— — —</td>
<td>— — —</td>
<td>RST 2</td>
</tr>
<tr>
<td>R104</td>
<td>2010.6 11 1401 311</td>
<td>— — —</td>
<td>— — —</td>
<td>RST 2</td>
</tr>
<tr>
<td>R105</td>
<td>2418.0 16 560 93</td>
<td>— — —</td>
<td>— — —</td>
<td>RST 1</td>
</tr>
<tr>
<td>R106</td>
<td>2256.9 14 1403 252</td>
<td>— — —</td>
<td>— — —</td>
<td>LOT</td>
</tr>
<tr>
<td>R107</td>
<td>2091.6 12 1462 278</td>
<td>— — —</td>
<td>— — —</td>
<td>LOT</td>
</tr>
<tr>
<td>R108</td>
<td>1980.3 10 2325 491</td>
<td>— — —</td>
<td>— — —</td>
<td>RST 0</td>
</tr>
<tr>
<td>R109</td>
<td>2191.4 13 2149 398</td>
<td>— — —</td>
<td>— — —</td>
<td>RST 1</td>
</tr>
<tr>
<td>R110</td>
<td>2121.1 12 1479 291</td>
<td>— — —</td>
<td>— — —</td>
<td>RST 2</td>
</tr>
<tr>
<td>R111</td>
<td>2082.1 12 1882 370</td>
<td>— — —</td>
<td>— — —</td>
<td>RST 2</td>
</tr>
<tr>
<td>R112</td>
<td>1986.1 11 2325 507</td>
<td>— — —</td>
<td>— — —</td>
<td>RST 1</td>
</tr>
<tr>
<td>C101</td>
<td>9827.3 10 285 45</td>
<td>9827.3 10 434.5</td>
<td>0.0 0.00%</td>
<td>OST</td>
</tr>
<tr>
<td>C102</td>
<td>9820.3 10 237 42</td>
<td>— — —</td>
<td>— — —</td>
<td>OST</td>
</tr>
<tr>
<td>C103</td>
<td>9813.7 10 256 49</td>
<td>— — —</td>
<td>— — —</td>
<td>OST</td>
</tr>
<tr>
<td>C104</td>
<td>9809.0 10 2495 536</td>
<td>— — —</td>
<td>— — —</td>
<td>RST 2</td>
</tr>
<tr>
<td>C105</td>
<td>9821.2 10 313 50</td>
<td>— — —</td>
<td>— — —</td>
<td>OST</td>
</tr>
<tr>
<td>C106</td>
<td>9827.3 10 455 75</td>
<td>9827.3 10 724.8</td>
<td>0.0 0.00%</td>
<td>OST</td>
</tr>
<tr>
<td>C107</td>
<td>9818.9 10 292 48</td>
<td>— — —</td>
<td>— — —</td>
<td>OST</td>
</tr>
<tr>
<td>C108</td>
<td>9818.9 10 662 115</td>
<td>— — —</td>
<td>— — —</td>
<td>OST</td>
</tr>
<tr>
<td>C109</td>
<td>9818.6 10 1381 262</td>
<td>— — —</td>
<td>— — —</td>
<td>LOT</td>
</tr>
<tr>
<td>RC101</td>
<td>2685.7 16 897 144</td>
<td>— — —</td>
<td>— — —</td>
<td>OST</td>
</tr>
<tr>
<td>RC102</td>
<td>2534.0 15 2410 434</td>
<td>— — —</td>
<td>— — —</td>
<td>OST</td>
</tr>
<tr>
<td>RC103</td>
<td>2352.3 13 1047 195</td>
<td>— — —</td>
<td>— — —</td>
<td>RST 0</td>
</tr>
<tr>
<td>RC104</td>
<td>2209.1 11 1311 272</td>
<td>— — —</td>
<td>— — —</td>
<td>RST 2</td>
</tr>
<tr>
<td>RC105</td>
<td>2538.0 15 2327 412</td>
<td>— — —</td>
<td>— — —</td>
<td>RST 1</td>
</tr>
<tr>
<td>RC106</td>
<td>2457.8 14 443 74</td>
<td>— — —</td>
<td>— — —</td>
<td>RST 0</td>
</tr>
<tr>
<td>RC107</td>
<td>2236.9 12 1822 344</td>
<td>— — —</td>
<td>— — —</td>
<td>RST 0</td>
</tr>
<tr>
<td>RC108</td>
<td>2115.9 11 2206 451</td>
<td>— — —</td>
<td>— — —</td>
<td>RST 1</td>
</tr>
<tr>
<td>Average</td>
<td>4624.9 12.45 1365 261.48</td>
<td>— — —</td>
<td>— — —</td>
<td>— — —</td>
</tr>
</tbody>
</table>

(Ryer 1999)

1 Maximum number of vehicles: $m = 25$. Time window penalty: $p_{tw} = 8.0$.
2 Maximum iterations: $k = 2500$.
3 Seconds on a Pentium II 400 MHz system. Total runtime ~ 550 seconds each.
4 Seconds on a Sun Sparc 1 workstation.
5 OST is ordered starting tour. RST is random starting tour seeded with the value given. LOT is listed ordering.
<table>
<thead>
<tr>
<th>Problem Set</th>
<th>O'Rourke &amp; Ryer</th>
<th>Optimal</th>
<th>Difference</th>
<th>Start Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Z(t)</td>
<td>Used</td>
<td>Iter</td>
<td>Time</td>
</tr>
<tr>
<td>R101</td>
<td>867.1</td>
<td>8</td>
<td>317</td>
<td>4</td>
</tr>
<tr>
<td>R102</td>
<td>797.1</td>
<td>7</td>
<td>35</td>
<td>1</td>
</tr>
<tr>
<td>R103</td>
<td>704.6</td>
<td>5</td>
<td>132</td>
<td>1</td>
</tr>
<tr>
<td>R104</td>
<td>666.9</td>
<td>4</td>
<td>86</td>
<td>2</td>
</tr>
<tr>
<td>R105</td>
<td>780.5</td>
<td>6</td>
<td>95</td>
<td>1</td>
</tr>
<tr>
<td>R106</td>
<td>715.4</td>
<td>5</td>
<td>1149</td>
<td>12</td>
</tr>
<tr>
<td>R107</td>
<td>674.3</td>
<td>4</td>
<td>2080</td>
<td>24</td>
</tr>
<tr>
<td>R108</td>
<td>647.3</td>
<td>4</td>
<td>58</td>
<td>1</td>
</tr>
<tr>
<td>R109</td>
<td>691.3</td>
<td>5</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>R110</td>
<td>694.1</td>
<td>5</td>
<td>91</td>
<td>1</td>
</tr>
<tr>
<td>R111</td>
<td>678.8</td>
<td>4</td>
<td>178</td>
<td>3</td>
</tr>
<tr>
<td>R112</td>
<td>643.0</td>
<td>4</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>C101</td>
<td>2441.3</td>
<td>3</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>C102</td>
<td>2440.3</td>
<td>3</td>
<td>106</td>
<td>1</td>
</tr>
<tr>
<td>C103</td>
<td>2440.3</td>
<td>3</td>
<td>72</td>
<td>1</td>
</tr>
<tr>
<td>C104</td>
<td>2436.9</td>
<td>3</td>
<td>741</td>
<td>8</td>
</tr>
<tr>
<td>C105</td>
<td>2441.3</td>
<td>3</td>
<td>170</td>
<td>1</td>
</tr>
<tr>
<td>C106</td>
<td>2441.3</td>
<td>3</td>
<td>35</td>
<td>1</td>
</tr>
<tr>
<td>C107</td>
<td>2441.3</td>
<td>3</td>
<td>51</td>
<td>0</td>
</tr>
<tr>
<td>C108</td>
<td>2441.3</td>
<td>3</td>
<td>455</td>
<td>4</td>
</tr>
<tr>
<td>C109</td>
<td>2441.3</td>
<td>3</td>
<td>197</td>
<td>2</td>
</tr>
<tr>
<td>RC101</td>
<td>711.1</td>
<td>4</td>
<td>214</td>
<td>2</td>
</tr>
<tr>
<td>RC102</td>
<td>601.7</td>
<td>3</td>
<td>149</td>
<td>1</td>
</tr>
<tr>
<td>RC103</td>
<td>582.8</td>
<td>3</td>
<td>134</td>
<td>2</td>
</tr>
<tr>
<td>RC104</td>
<td>556.6</td>
<td>3</td>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>RC105</td>
<td>661.2</td>
<td>4</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>RC106</td>
<td>595.5</td>
<td>3</td>
<td>60</td>
<td>1</td>
</tr>
<tr>
<td>RC107</td>
<td>548.3</td>
<td>3</td>
<td>179</td>
<td>2</td>
</tr>
<tr>
<td>RC108</td>
<td>544.5</td>
<td>3</td>
<td>353</td>
<td>3</td>
</tr>
<tr>
<td>Average</td>
<td>1218.2</td>
<td>3.93</td>
<td>250.7</td>
<td>2.86</td>
</tr>
</tbody>
</table>

(1) Maximum number of vehicles: \( m = 10 \). Time window penalty: \( \rho_{tw} = 8.0 \); load penalty \( \rho_{ld} = 10.0 \).

(2) Maximum iterations: \( k = 2500 \).

(3) Seconds on a Pentium II 400 MHz system. Total runtime ~ 28 seconds each.

(4) Seconds on a Sun Sparc 1 workstation.

(5) OST is ordered starting tour. RST is random starting tour seeded with the value given. LOT is listed ordering.

(Ryer 1999)
## Table 5. Solomon VRPTW Computational Results (50 Customers)

<table>
<thead>
<tr>
<th>Problem Set</th>
<th>O'Rourke &amp; Ryan</th>
<th></th>
<th></th>
<th>Optimal</th>
<th></th>
<th></th>
<th>Difference</th>
<th>Start Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Z(t)</td>
<td>Used</td>
<td>Iter²</td>
<td>Time³</td>
<td>Z(t)</td>
<td>Used</td>
<td>Time⁴</td>
<td>Δ</td>
</tr>
<tr>
<td>R101</td>
<td>1543.8</td>
<td>12</td>
<td>239</td>
<td>9</td>
<td>1535.2</td>
<td>12</td>
<td>66.7</td>
<td>8.6</td>
</tr>
<tr>
<td>R102</td>
<td>1409.0</td>
<td>11</td>
<td>1939</td>
<td>82</td>
<td>1404.6</td>
<td>11</td>
<td>67.8</td>
<td>4.4</td>
</tr>
<tr>
<td>R103</td>
<td>1278.7</td>
<td>9</td>
<td>1935</td>
<td>87</td>
<td>1272.5</td>
<td>9</td>
<td>8939.1</td>
<td>6.2</td>
</tr>
<tr>
<td>R104</td>
<td>1137.4</td>
<td>6</td>
<td>1533</td>
<td>69</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R105</td>
<td>1401.6</td>
<td>9</td>
<td>402</td>
<td>16</td>
<td>1399.2</td>
<td>9</td>
<td>362.6</td>
<td>2.4</td>
</tr>
<tr>
<td>R106</td>
<td>1293.0</td>
<td>8</td>
<td>2294</td>
<td>99</td>
<td>1285.2</td>
<td>8</td>
<td>386.4</td>
<td>7.8</td>
</tr>
<tr>
<td>R107</td>
<td>1211.1</td>
<td>7</td>
<td>1786</td>
<td>79</td>
<td>1211.1</td>
<td>7</td>
<td>7362.1</td>
<td>0.0</td>
</tr>
<tr>
<td>R108</td>
<td>1117.7</td>
<td>6</td>
<td>1698</td>
<td>78</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R109</td>
<td>1286.7</td>
<td>8</td>
<td>1451</td>
<td>61</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R110</td>
<td>1207.8</td>
<td>7</td>
<td>1853</td>
<td>84</td>
<td>1197.0</td>
<td>7</td>
<td>4906.1</td>
<td>10.8</td>
</tr>
<tr>
<td>R111</td>
<td>1216.6</td>
<td>7</td>
<td>1775</td>
<td>76</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R112</td>
<td>1135.0</td>
<td>6</td>
<td>1456</td>
<td>68</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>C101</td>
<td>4862.4</td>
<td>5</td>
<td>74</td>
<td>3</td>
<td>4862.4</td>
<td>5</td>
<td>67.1</td>
<td>0.0</td>
</tr>
<tr>
<td>C102</td>
<td>4861.4</td>
<td>5</td>
<td>232</td>
<td>9</td>
<td>4861.4</td>
<td>5</td>
<td>330.2</td>
<td>0.0</td>
</tr>
<tr>
<td>C103</td>
<td>4861.4</td>
<td>5</td>
<td>2035</td>
<td>87</td>
<td>4861.4</td>
<td>5</td>
<td>896.0</td>
<td>0.0</td>
</tr>
<tr>
<td>C104</td>
<td>4882.8</td>
<td>5</td>
<td>1727</td>
<td>79</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>C105</td>
<td>4862.4</td>
<td>5</td>
<td>494</td>
<td>19</td>
<td>4862.4</td>
<td>5</td>
<td>99.1</td>
<td>0.0</td>
</tr>
<tr>
<td>C106</td>
<td>4862.4</td>
<td>5</td>
<td>91</td>
<td>4</td>
<td>4862.4</td>
<td>5</td>
<td>91.3</td>
<td>0.0</td>
</tr>
<tr>
<td>C107</td>
<td>4862.4</td>
<td>5</td>
<td>154</td>
<td>6</td>
<td>4862.4</td>
<td>5</td>
<td>170.6</td>
<td>0.0</td>
</tr>
<tr>
<td>C108</td>
<td>4862.4</td>
<td>5</td>
<td>95</td>
<td>4</td>
<td>4862.4</td>
<td>5</td>
<td>245.6</td>
<td>0.0</td>
</tr>
<tr>
<td>C109</td>
<td>4862.4</td>
<td>5</td>
<td>643</td>
<td>26</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RC101</td>
<td>1446.8</td>
<td>8</td>
<td>1613</td>
<td>60</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RC102</td>
<td>1331.8</td>
<td>7</td>
<td>1508</td>
<td>60</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RC103</td>
<td>1210.9</td>
<td>6</td>
<td>2194</td>
<td>94</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RC104</td>
<td>1046.5</td>
<td>5</td>
<td>412</td>
<td>18</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RC105</td>
<td>1355.3</td>
<td>8</td>
<td>104</td>
<td>4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RC106</td>
<td>1223.2</td>
<td>6</td>
<td>1454</td>
<td>58</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RC107</td>
<td>1144.4</td>
<td>6</td>
<td>898</td>
<td>36</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RC108</td>
<td>1098.1</td>
<td>6</td>
<td>1361</td>
<td>58</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Average</td>
<td>2375.01</td>
<td>6.66</td>
<td>1153</td>
<td>49.4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

(Ryer 1999)

1. Maximum number of vehicles: $m = 15$. Time window penalty: $\rho_{TW} = 1.0$; load penalty $\rho_{LO} = 10.0$.
2. Maximum iterations $k = 2500$.
3. Seconds on a Pentium II 400 MHz system. Total runtime ~ 100 seconds each.
4. Seconds on a Sun Sparc 1 workstation.

OST is ordered starting tour. RST is random starting tour seeded with the value given. LOT is listed ordering.
<table>
<thead>
<tr>
<th>Problem Set</th>
<th>O'Rourke &amp; Ryer</th>
<th>Optimal</th>
<th>Difference</th>
<th>Start Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Z(t) Used Iter</td>
<td>Z(t) Used Time</td>
<td>Δ</td>
<td>Δ%</td>
</tr>
<tr>
<td>R101</td>
<td>2676.2 20 2271 414</td>
<td>2607.7 18 1064.2</td>
<td>68.5</td>
<td>2.63%</td>
</tr>
<tr>
<td>R102</td>
<td>2502.4 19 492 96</td>
<td>2434.0 17 756.9</td>
<td>68.4</td>
<td>2.81%</td>
</tr>
<tr>
<td>R103</td>
<td>2265.0 15 1091 228</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R104</td>
<td>2039.6 12 1488 338</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R105</td>
<td>2399.4 16 1974 378</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R106</td>
<td>2268.4 14 2431 491</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R107</td>
<td>2129.0 13 1905 406</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R108</td>
<td>1956.8 10 2415 565</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R109</td>
<td>2181.0 14 1587 311</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R110</td>
<td>2133.2 13 1548 328</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R111</td>
<td>2077.3 12 2248 491</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>R112</td>
<td>1971.6 11 1898 460</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>C101</td>
<td>9827.3 10 263 43</td>
<td>9827.3 10 434.5</td>
<td>0.0</td>
<td>0.00%</td>
</tr>
<tr>
<td>C102</td>
<td>9827.3 10 1317 253</td>
<td>9827.3 10 1990.8</td>
<td>0.0</td>
<td>0.00%</td>
</tr>
<tr>
<td>C103</td>
<td>9828.9 10 2500 535</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>C104</td>
<td>9949.6 10 2194 500</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>C105</td>
<td>9827.3 10 378 65</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>C106</td>
<td>9827.3 10 309 55</td>
<td>9827.3 10 724.8</td>
<td>0.0</td>
<td>0.00%</td>
</tr>
<tr>
<td>C107</td>
<td>9827.3 10 1144 210</td>
<td>9827.3 10 1010.4</td>
<td>0.0</td>
<td>0.00%</td>
</tr>
<tr>
<td>C108</td>
<td>9827.3 10 1638 321</td>
<td>9827.3 10 1613.6</td>
<td>0.0</td>
<td>0.00%</td>
</tr>
<tr>
<td>C109</td>
<td>9833.3 10 2202 463</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RC101</td>
<td>2669.9 16 2110 381</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RC102</td>
<td>2498.4 15 2136 419</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RC103</td>
<td>2363.6 13 1333 270</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RC104</td>
<td>2179.2 11 1365 308</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RC105</td>
<td>2357.4 15 2482 473</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RC106</td>
<td>2432.8 13 2222 434</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RC107</td>
<td>2266.1 12 2024 417</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RC108</td>
<td>2175.1 12 2122 475</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Average</td>
<td>4632.3 12.62 1693 349.6</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

(Ryer 1999)  

1 Maximum number of vehicles: m = 25. Time window penalty: \( \rho_{tw} = 8.0 \); load penalty \( \rho_{ld} = 10.0 \).  
2 Maximum iterations \( k = 2500 \).  
3 Seconds on a Pentium II 400 MHz system. Total runtime \( \sim 550 \) seconds each.  
4 Seconds on a Sun Sparc 1 workstation.  
5 OST is ordered starting tour. RST is random starting tour seeded with the value given. LOT is listed ordering.
2.4.2 UAV Results.

We analyzed a Bosnia UAV scenario provided by Bergdahl (1998). Winds for the region of interest are given in Table 7. These winds are taken from actual US Air Force meteorological conditions for the operating region.

Table 7. Wind Data

<table>
<thead>
<tr>
<th>Altitude Tier</th>
<th>Altitude (ft)</th>
<th>$\Theta_{WS}$ (deg)</th>
<th>$WS$ (kts)</th>
<th>$AS$ (kts)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5,000</td>
<td>300</td>
<td>7.5</td>
<td>70</td>
</tr>
<tr>
<td>1</td>
<td>10,000</td>
<td>300</td>
<td>37.5</td>
<td>70</td>
</tr>
<tr>
<td>2</td>
<td>18,000</td>
<td>310</td>
<td>50.0</td>
<td>70</td>
</tr>
</tbody>
</table>

Scenario details are listed in Table 8, and a map of this scenario is provided in Figure 9. The 52 targets fall into three remote operating zones (ROZs), each with non-overlapping time windows. Route optimization begins and ends with the Srbac, Bosnia waypoint, since the route to and from there must follow a mandatory air corridor.

The scenario was solved in 108 seconds on a Pentium II 300 MHz system using the UAV specific module of the heuristic. With optimum use of wind tiers, the solver returned a tour requiring only one vehicle with a mission time of 822 minutes. Without wind tier modeling, two vehicles are required with a combined mission time of 1384 minutes. This demonstrates the improvement that can be achieved with smart selection of travel altitudes.

The optimized tour output is listed in Table 9 (the "Alt" column designates the altitude tier to be used enroute to the next target); this flight path is shown in Figures...
Figure 10 and Figure 11. Figure 11 shows the same sequence as Figure 10 with a temporal component as the added third axis and gray bars representing the time windows.

2.5 Conclusions

Our Java implementation of a reactive tabu search first described by Battiti and Tecchiolli (1994) successfully solves single/multiple traveling salesman problems with and without time windows (TSP, MTSP, TSPTW, MTSTPW), as well as capacitated vehicle routing problems with and without time windows (VRP, VRPTW). On the Solomon problem sets, our heuristic produces close to optimal solutions within reasonable computing times. Addition of reactive penalties allows the algorithm to perform more robustly over a wider set of problems.

Our implementation supports UAV problems and formats as well as classical problems and formats. Changes required for the UAV problem reflect unique aspects of the operational UAV mission and include items such as a reformulated objective function, alternate coordinate and numerical formatting, and random customer service times. The introduction of altitude-based wind tiers, when selecting UAV routes, capitalizes on the altitude-dependent, highly asymmetric nature of the flight environment.

The Java implementation is an object oriented structure that is both machine portable and readily modifiable to support new problem instances. The internal data structure and methodology work with the GUI to support operational requirements such as route locking and dynamic rescheduling in support of priority targets.
Table 8. Bosnia Data Set

<table>
<thead>
<tr>
<th>Location Name</th>
<th>Lat (DD MM SS)</th>
<th>Lon (DD MM SS)</th>
<th>$e_i$</th>
<th>$\ell_i$</th>
<th>$s_{min(i)}$</th>
<th>$s_{max(i)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>DepotTzarBosnia</td>
<td>N 46 24 0</td>
<td>E 17 54 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CorridorSzulokBosnia</td>
<td>N 46 3 45</td>
<td>E 17 32 44</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CorridorSzulok</td>
<td>N 46 3 45</td>
<td>E 17 32 44</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CorridorSrbac</td>
<td>N 45 24 0</td>
<td>E 17 54 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dumdugja</td>
<td>N 44 58 29</td>
<td>E 16 50 34</td>
<td>0</td>
<td>4800</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mastyje</td>
<td>N 44 58 46</td>
<td>E 16 38 56</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>AAA Site Garred</td>
<td>N 44 58 4</td>
<td>E 16 39 31</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>Hvy Wpn Dep Tharmet</td>
<td>N 44 58 33</td>
<td>E 16 39 18</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>Hvy Wpn Dep Tharmet</td>
<td>N 44 58 39</td>
<td>E 16 39 41</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>Hvy Wpn Dep Tharmet</td>
<td>N 44 58 59</td>
<td>E 16 39 28</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>Comm Site Sardona</td>
<td>N 44 59 2</td>
<td>E 16 39 56</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>Comm Site Sardona</td>
<td>N 44 59 11</td>
<td>E 16 40 19</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>Comm Site Sardona</td>
<td>N 44 59 15</td>
<td>E 16 39 20</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>Susp Wpn Storage</td>
<td>N 44 59 9</td>
<td>E 16 39 10</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>Susp Wpn Storage</td>
<td>N 44 54 52</td>
<td>E 16 34 47</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>Susp Wpn Storage</td>
<td>N 44 51 49</td>
<td>E 16 41 37</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>Susp Wpn Storage</td>
<td>N 44 59 9</td>
<td>E 16 49 17</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>Susp Wpn Storage</td>
<td>N 44 57 41</td>
<td>E 16 39 35</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>SAM IAD Site Prob SA2</td>
<td>N 44 57 23</td>
<td>E 16 51 45</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>SAM IAD Site Prob SA2</td>
<td>N 44 57 45</td>
<td>E 16 49 28</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>SAM IAD Site Prob SA2</td>
<td>N 44 55 57</td>
<td>E 16 43 52</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>SAM IAD Site Radar</td>
<td>N 44 57 47</td>
<td>E 16 39 54</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>HQ Site Dromada</td>
<td>N 45 0 7</td>
<td>E 16 53 49</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>Warehouse Dromada</td>
<td>N 44 53 31</td>
<td>E 16 54 12</td>
<td>1015</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>Barracks 1 Omannski</td>
<td>N 44 45 34</td>
<td>E 16 10 34</td>
<td>1500</td>
<td>5</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>Barracks 1 Omannski</td>
<td>N 44 58 19</td>
<td>E 16 12 14</td>
<td>1500</td>
<td>5</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>Barracks 1 Omannski</td>
<td>N 44 51 2</td>
<td>E 17 13 24</td>
<td>1500</td>
<td>5</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>Tank Rally Point Bolstavec</td>
<td>N 44 50 51</td>
<td>E 17 14 39</td>
<td>1500</td>
<td>5</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>Tank Rally Point Bolstavec</td>
<td>N 44 56 17</td>
<td>E 17 17 41</td>
<td>1500</td>
<td>5</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>Storage Bunker Krajachastane</td>
<td>N 44 55 51</td>
<td>E 17 17 51</td>
<td>1500</td>
<td>2</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Storage Bunker Krajachastane</td>
<td>N 44 56 7</td>
<td>E 17 18 23</td>
<td>1500</td>
<td>2</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Road Golatrany</td>
<td>N 44 28 13</td>
<td>E 17 1 1</td>
<td>1730</td>
<td>20</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>Road Golatrany</td>
<td>N 44 27 29</td>
<td>E 17 1 46</td>
<td>1730</td>
<td>20</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>Road Golatrany</td>
<td>N 44 27 10</td>
<td>E 17 1 24</td>
<td>1730</td>
<td>20</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>Dumdugja</td>
<td>N 44 58 29</td>
<td>E 16 50 34</td>
<td>1900</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>Mastye</td>
<td>N 44 58 46</td>
<td>E 16 38 56</td>
<td>1900</td>
<td>30</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>AAA Site Garred</td>
<td>N 44 58 4</td>
<td>E 16 39 31</td>
<td>1900</td>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Hvy Wpn Dep Tharmet</td>
<td>N 44 58 33</td>
<td>E 16 39 18</td>
<td>1900</td>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Hvy Wpn Dep Tharmet</td>
<td>N 44 58 39</td>
<td>E 16 39 41</td>
<td>1900</td>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Hvy Wpn Dep Tharmet</td>
<td>N 44 58 59</td>
<td>E 16 39 28</td>
<td>1900</td>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Comm Site Sardona</td>
<td>N 44 59 2</td>
<td>E 16 39 56</td>
<td>1900</td>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Comm Site Sardona</td>
<td>N 44 59 11</td>
<td>E 16 40 19</td>
<td>1900</td>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Comm Site Sardona</td>
<td>N 44 59 15</td>
<td>E 16 39 20</td>
<td>1900</td>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Susp Wpn Storage</td>
<td>N 44 59 9</td>
<td>E 16 49 17</td>
<td>1900</td>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Susp Wpn Storage</td>
<td>N 44 57 41</td>
<td>E 16 39 35</td>
<td>1900</td>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>SAM IAD Site Prob SA2</td>
<td>N 44 57 23</td>
<td>E 16 51 45</td>
<td>1900</td>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>SAM IAD Site Prob SA2</td>
<td>N 44 57 45</td>
<td>E 16 49 28</td>
<td>1900</td>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>SAM IAD Site Prob SA2</td>
<td>N 44 55 57</td>
<td>E 16 43 52</td>
<td>1900</td>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>SAM IAD Site Radar</td>
<td>N 44 57 47</td>
<td>E 16 39 54</td>
<td>1900</td>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>HQ Site Dromada</td>
<td>N 45 0 7</td>
<td>E 16 53 49</td>
<td>1900</td>
<td>30</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>Warehouse Dromada</td>
<td>N 44 53 31</td>
<td>E 16 54 12</td>
<td>1900</td>
<td>2</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>Corridor Srbac</td>
<td>N 45 24 0</td>
<td>E 17 30 0</td>
<td>940</td>
<td>0</td>
<td>4740</td>
<td></td>
</tr>
</tbody>
</table>

1 Time listed in hours-minutes format.
2 Minutes.
* Optimization begins from Srbac Corridor waypoint

(Bergdahl 1998)
Figure 9. Bosnia Scenario Target Locations
Table 9. Bosnia Tour Sequence

<table>
<thead>
<tr>
<th>Label</th>
<th>ID</th>
<th>Lat(^a)</th>
<th>Long(^b)</th>
<th>Early(^c)</th>
<th>Late(^d)</th>
<th>Arr(^d)</th>
<th>Dep(^d)</th>
<th>Serv(^d)</th>
<th>Alt(^d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corridor Srebrenica</td>
<td>0</td>
<td>-17.5416</td>
<td>580</td>
<td>2880</td>
<td>580.00</td>
<td>580.0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>HQ Site Dramada</td>
<td>20</td>
<td>-16.8969</td>
<td>615</td>
<td>900</td>
<td>606.25</td>
<td>615.0</td>
<td>30</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>SAMIAD Site ProbSA2</td>
<td>16</td>
<td>-16.8625</td>
<td>615</td>
<td>900</td>
<td>647.66</td>
<td>647.6</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Dumduga</td>
<td>1</td>
<td>-16.8427</td>
<td>615</td>
<td>900</td>
<td>650.97</td>
<td>650.9</td>
<td>125</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Susp Wpn Storage</td>
<td>14</td>
<td>-16.8213</td>
<td>615</td>
<td>900</td>
<td>778.02</td>
<td>778.0</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SAMIAD Site ProbSA2</td>
<td>17</td>
<td>-16.8244</td>
<td>615</td>
<td>900</td>
<td>780.89</td>
<td>780.8</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SAMIAD Site ProbSA2</td>
<td>18</td>
<td>-16.7311</td>
<td>615</td>
<td>900</td>
<td>786.88</td>
<td>786.8</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Comm Site Sardona</td>
<td>8</td>
<td>-16.6719</td>
<td>615</td>
<td>900</td>
<td>792.77</td>
<td>792.7</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Comm Site Sardona</td>
<td>7</td>
<td>-16.6655</td>
<td>615</td>
<td>900</td>
<td>795.05</td>
<td>795.0</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Comm Site Sardona</td>
<td>9</td>
<td>-16.6555</td>
<td>615</td>
<td>900</td>
<td>797.50</td>
<td>797.5</td>
<td>6</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Susp Wpn Storage</td>
<td>10</td>
<td>-16.6527</td>
<td>615</td>
<td>900</td>
<td>804.11</td>
<td>804.1</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Hwy Wpn Dep Tharmet</td>
<td>6</td>
<td>-16.6577</td>
<td>615</td>
<td>900</td>
<td>807.86</td>
<td>807.8</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Hwy Wpn Dep Tharmet</td>
<td>5</td>
<td>-16.6613</td>
<td>615</td>
<td>900</td>
<td>810.05</td>
<td>810.0</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SAMIAD Site Radar</td>
<td>19</td>
<td>-16.6685</td>
<td>615</td>
<td>900</td>
<td>812.57</td>
<td>812.5</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Susp Wpn Storage</td>
<td>12</td>
<td>-16.6547</td>
<td>615</td>
<td>900</td>
<td>814.79</td>
<td>814.7</td>
<td>6</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Warehouse Dramada</td>
<td>21</td>
<td>-16.6597</td>
<td>615</td>
<td>900</td>
<td>817.14</td>
<td>817.1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Tank Rally Point Bstevac</td>
<td>26</td>
<td>-16.6536</td>
<td>615</td>
<td>900</td>
<td>821.93</td>
<td>821.9</td>
<td>30</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Storage Bunker Krajachstan</td>
<td>28</td>
<td>-17.3063</td>
<td>900</td>
<td>1035</td>
<td>905.98</td>
<td>905.9</td>
<td>15</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Storage Bunker Krajachstan</td>
<td>27</td>
<td>-17.2975</td>
<td>900</td>
<td>1035</td>
<td>921.88</td>
<td>921.8</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Tank Rally Point Bstevac</td>
<td>25</td>
<td>-17.2441</td>
<td>900</td>
<td>1035</td>
<td>928.56</td>
<td>928.5</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>barracks Omasnki</td>
<td>24</td>
<td>-17.2333</td>
<td>900</td>
<td>1035</td>
<td>931.42</td>
<td>931.4</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>barracks Omasnki</td>
<td>23</td>
<td>-17.2038</td>
<td>900</td>
<td>1035</td>
<td>949.23</td>
<td>949.2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>barracks Omasnki</td>
<td>22</td>
<td>-17.1761</td>
<td>900</td>
<td>1035</td>
<td>956.77</td>
<td>956.7</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Road Golubovici</td>
<td>31</td>
<td>-17.04</td>
<td>1050</td>
<td>1110</td>
<td>977.93</td>
<td>977.9</td>
<td>20</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Road Golubovici</td>
<td>30</td>
<td>-17.0294</td>
<td>1050</td>
<td>1110</td>
<td>1070.52</td>
<td>1070.5</td>
<td>20</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Road Golubovici</td>
<td>29</td>
<td>-17.0216</td>
<td>1050</td>
<td>1110</td>
<td>1091.27</td>
<td>1091.2</td>
<td>20</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Susp Wpn Storage</td>
<td>45</td>
<td>-16.6936</td>
<td>1140</td>
<td>1380</td>
<td>1139.59</td>
<td>1139.5</td>
<td>0</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Susp Wpn Storage</td>
<td>42</td>
<td>-16.5979</td>
<td>1140</td>
<td>1380</td>
<td>1228.84</td>
<td>1228.8</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Susp Wpn Storage</td>
<td>44</td>
<td>-16.5797</td>
<td>1140</td>
<td>1380</td>
<td>1284.84</td>
<td>1284.8</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Martys</td>
<td>33</td>
<td>-16.6488</td>
<td>1140</td>
<td>1380</td>
<td>1169.55</td>
<td>1169.5</td>
<td>0</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Susp Wpn Storage</td>
<td>41</td>
<td>-16.6527</td>
<td>1140</td>
<td>1380</td>
<td>1199.91</td>
<td>1199.9</td>
<td>24</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Comm Site Sardona</td>
<td>40</td>
<td>-16.6555</td>
<td>1140</td>
<td>1380</td>
<td>1224.17</td>
<td>1224.1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Comm Site Sardona</td>
<td>39</td>
<td>-16.6719</td>
<td>1140</td>
<td>1380</td>
<td>1226.56</td>
<td>1226.5</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Comm Site Sardona</td>
<td>38</td>
<td>-16.6655</td>
<td>1140</td>
<td>1380</td>
<td>1228.84</td>
<td>1228.8</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Hwy Wpn Dep Tharmet</td>
<td>37</td>
<td>-16.6577</td>
<td>1140</td>
<td>1380</td>
<td>1284.84</td>
<td>1284.8</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Hwy Wpn Dep Tharmet</td>
<td>36</td>
<td>-16.6613</td>
<td>1140</td>
<td>1380</td>
<td>1284.55</td>
<td>1284.5</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Hwy Wpn Dep Tharmet</td>
<td>35</td>
<td>-16.6549</td>
<td>1140</td>
<td>1380</td>
<td>1288.13</td>
<td>1288.1</td>
<td>28</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>AAA Site Carred</td>
<td>34</td>
<td>-16.6586</td>
<td>1140</td>
<td>1380</td>
<td>1318.38</td>
<td>1318.3</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Susp Wpn Storage</td>
<td>46</td>
<td>-16.6597</td>
<td>1140</td>
<td>1380</td>
<td>1320.94</td>
<td>1320.9</td>
<td>30</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SAMIAD Site Radar</td>
<td>20</td>
<td>-16.6657</td>
<td>1140</td>
<td>1380</td>
<td>1353.14</td>
<td>1353.1</td>
<td>30</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>SAMIAD Site ProbSA2</td>
<td>49</td>
<td>-16.6532</td>
<td>1140</td>
<td>1380</td>
<td>1401.57</td>
<td>-------</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^a\) Parameters set as follows: maximum number of vehicles: \( m = 5 \); maximum iterations: \( k = 2500 \); reactive penalty scheme; LOT starting tour. Total runtime 108 seconds on a Pentium II 300 MHz system.

\(^b\) By convention, north latitudes are positive and south latitudes are negative.

\(^c\) By convention, west longitudes are positive and east longitudes are negative.

\(^d\) Time in minutes.

\(^e\) Flight altitude to next point: "0" = 5,000 ft, "1" = 10,000 ft, "2" = 18,000 ft.
Figure 11. Temporal Route Plot
Chapter 3

We present several ideas that represent natural and worthwhile extensions to the work accomplished.

3.1 Heuristic Modifications

Modifications to the tabu search heuristic could include any of the following ideas. Additional operators have increased solution quality for genetic algorithms (Petridis et al. 1998); construction and implementation of additional operators may prove useful. These operators could consider additional random or directed moves which expand the neighborhood, such as a 4-opt, for possible improvements in the objective function.

Restarts based on changes in the solution quality or stabilization of the objective function could prove useful. Methods to consider include the following: maintenance of an elite list of best solutions where a restart resumes with relaxed tabu restrictions (Xu and Kelly 1996); intensification from previous location with stored tabu status (Armentano and Ronconi 1999); or a multi-start backjump tracking scheme (Liaw 1999, Norwicki and Smutnicki 1996). Other initialization methods such as a sweep initialization or petal initialization (Renaud 1996b) could be explored.
3.2 UAV Related Modifications

Changes to the UAV specific aspect of the problem could include a priority scheme hierarchy that generates route segments based on assigned target priorities. This would involve constructing subtours that are then smartly linked together—obviously, the parameters of one subtour will be highly dependent on the others. The rudimentary wind modeling (discrete levels and average regional values) could be replaced with wind values that correlate specifically to each leg. A more detailed modeling of actual UAV transition times between altitudes, with modeled climb rates would provide a higher fidelity mission profile. The service times distribution model, which is still rather unknown, could be updated to reflect data gathered from recent operations.

3.3 Java Code Modifications

Although we are not strict computer programmers, work was done in an attempt to improve the code for better heuristic performance. This includes items such as ordering logic comparisons (so that the most likely outcome is encountered first to reduce comparisons) and changing several methods (to decrease instantiations). These optimization modifications reduced the run time for the 100 customer, 25 vehicle Solomon problem sets from an average of 700 seconds to 550 seconds. While this represents about a 25% reduction in run time, there is still tremendous room for improvement due to excessive object copying.

With any Java non-primitive type, the statement “x = y” will cause the “x” label to point to the “y” object, and the previous “x” object (if any) will be lost. What remains is
the “y” object with both an “x” label and a “y” label. In order to have a separate “x”
object that is the same as the “y” object, an explicit copy or clone function must be used
to duplicate the object (Flanagan 1997). This duplication is an expensive operation, as it
instantiates a new object and copies the member data. Analysis of the current reactive
tabu heuristic with the KL Group’s JProbe™ Java profiler tool revealed that nearly 50%
of the run time is spent copying NodeType objects. Initial experimentation using an
index system as node pointers showed potential run times that are only 20% of the
current run time—100 customer, 25 vehicle Solomon sets ran in ~120 seconds versus the
current ~550 seconds. This speed increase results from copying and manipulating the
indices, which are Java primitive types, instead of copying and manipulating the
NodeType objects.

Some initial work was done in an attempt to reconfigure the heuristic to run using
indices, but the changes are substantial as they touch nearly every aspect of the program.
Future programming efforts should make the conversion, which will allow faster
solutions and larger problem sets.
Bibliography


Appendix A. Extended Problem Formulation

This appendix examines the formulation of the traveling salesman problem (TSP), multiple traveling salesman problem (MTSP), vehicle routing problem (VRP), and multiple-depot vehicle routing problem (MDVRP). It is provided for generality and thoroughness as these problem types have additional constraints which were not mentioned previously since they are intrinsically modeled in the tabu search heuristic. For instance, based on the way the tabu search evaluates the neighborhood and swaps customers, no subtour breaking constraint is provided since it is impossible for the heuristic to construct a subtour. Notation and numbering of variables differs slightly from that presented in Chapter 2, as the notation there is tailored to the problem context.

A.1 Traveling Salesman Problem (TSP)

The first problem class, and basis for the remaining types, is the traveling salesman problem (TSP). Begin by defining the TSP structure and objective as follows: Let $G$ be our network with the set of nodes $N$, a set of branches $A$, and the associated non-negative branch costs of $C = c_{ij}$. The objective of this problem is to form a tour spanning all the nodes beginning and ending at the origin (node 1), which yields the minimum total tour length or cost. In the most basic case, we assume that the costs are symmetric ($c_{ij} = c_{ji}$), but the problem can be asymmetric with no loss of generality.
This can be represented as an assignment problem, where exactly one arc \( x_{ij} \) starts at node \( i \), and exactly one arc \( x_{ij} \) terminates at node \( j \). Specifically, the problem is formulated as follows:

Minimize \( Z(t) = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \) \hspace{1cm} (A1.1)

Where

\[
    x_{ij} = \begin{cases} 
    1 & \text{if arc } ij \text{ is in the tour} \\
    0 & \text{otherwise}
    \end{cases}
\]

Subject to:

\[
    \sum_{i=1}^{n} x_{ij} = b_{j} = 1 \quad (j=1,2,\ldots,n) \] \hspace{1cm} (A1.2)

\[
    \sum_{j=1}^{n} x_{ij} = a_{i} = 1 \quad (i=1,2,\ldots,n) \] \hspace{1cm} (A1.3)

Where

\[ X = \{x_{ij}\} \in S, \quad x_{ij} \in \{0,1\} \quad \forall \ i, j = 1,2,\ldots,n. \]

As previously mentioned, additional constraints are required to eliminate subtours. Adding the subtour breaking constraint to the assignment formulation prevents subtours. The three standard ways to represent a subtour breaking constraint (Bodin et al. 1983) are listed as follows:

1. \( S = \left\{ (x_{ij}) : \sum_{i \in Q} \sum_{j \notin Q} x_{ij} \geq 1 \text{ for every nonempty proper subset } Q \text{ of } N \right\} \)

2. \( S = \left\{ (x_{ij}) : \sum_{i \in R} \sum_{j \in R} x_{ij} \leq |R| - 1 \text{ for every nonempty subset } R \text{ of } \{2,3,\ldots,n\} \right\} \)

3. \( S = \left\{ (x_{ij}) : y_{i} - y_{j} + nx_{ij} \leq n - 1 \text{ for } 2 \leq i \neq j \leq n \text{ for some real numbers } y_{i} \right\} \).
The first constraint requires that every node subset $Q$ of the solution set be connected to all of the other nodes in the solution. The second constraint requires that the arcs in the solution set contain no cycles (a cycle over $R$ nodes must contain $|R|$ arcs). The third constraint is not intuitively straightforward and calls for more explanation. First, define $y_i$ as follows:

$$y_i = \begin{cases} t & \text{if node } i \text{ is visited on the } i^{th} \text{ step in a tour} \\ 0 & \text{otherwise} \end{cases}$$

For an arc in the solution tour ($x_{ij} = 1$), the constraint becomes

$$t - (t + 1) + n \leq n - 1 .$$

For an arc not contained in the solution tour ($x_{ij} = 0$), the constraint reduces to

$$y_i - y_j \leq n - 1 .$$

The third representation has the advantage of adding only $n^2 - 3n + 2$ subtour breaking constraints to the formulation, where the previous two add $2^n$ constraints (Bodin et al. 1983).

### A.2 Multiple Traveling Salesman Problem (MTSP)

Adding more salesmen to the problem gives the next level of complexity, the multiple traveling salesman problem (MTSP). Let $m$ be the number of salesmen or vehicles that make up the fleet. Again the objective is to minimize the total distance traveled. Assume further that the $m$ salesmen depart from and return to the same depot and that each customer must be visited exactly once by exactly one salesman.
With these changes, the formulation is an extension of the basic TSP presented above and is represented as

Minimize \( Z(t) = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \) \hspace{1cm} (A2.1)

Subject to:

\[
\sum_{j=1}^{n} x_{ij} = b_j = \begin{cases} M & \text{if } j = 1 \\ 1 & \text{if } j = 2,3,\ldots,n \end{cases} \hspace{1cm} (A2.2)
\]

\[
\sum_{j=1}^{n} x_{ij} = a_j = \begin{cases} M & \text{if } j = 1 \\ 1 & \text{if } j = 2,3,\ldots,n \end{cases} \hspace{1cm} (A2.3)
\]

where \( X = (x_{ij}) \in S \) , \( x_{ij} \in \{0,1\} \) \( \forall i,j=1,2,\ldots,n \).

The first constraint in the formulation requires that all salesmen be used by forcing them to leave the depot. The second constraint requires all salesmen to return to the depot. Any one of the subtour breaking constraints used earlier in the TSP can be used for the MTSP.

The apparent complexity of this new problem can be reduced by representing the MTSP as \( m \) copies of the single TSP. This is accomplished by creating dummy depots \((D_1,\ldots,D_m)\) that are connected to the original network. These \( m \) copies are either separate from each other, or are connected with cost prohibitive big \( M \) arcs. When these single TSP copies are connected to a common depot, the problem becomes a series of \( m \) subtours, which when taken together forms the MTSP. This relatively straightforward transformation of the MTSP helps demonstrates why a TSP algorithm can be used to solve MTSP problems (Bodin et al. 1983).
A.3 Vehicle Routing Problem (VRP)

The next extension of the TSP is the Vehicle Routing Problem (VRP) which is obtained by adding a capacity constraint to the salesman or vehicles. In the VRP, a number of vehicles \( w \) leave a depot and service a number of customers \( n \), each possessing a unique demand \( d_i \). Each vehicle \( v \) has a limited capacity \( K_v \) and a maximum route duration \( T_v \) that constrains their closed delivery routes, or return to depot time. This particular instance of the VRP is commonly known as the general vehicle routing problem (GVRP). If the maximum route lengths or range constraints are removed, then this problem is referred to as the standard vehicle routing problem (SVRP) (Bodin et al. 1983). Additionally, the time required for a vehicle \( v \) to deliver or service at node \( i \) is \( s_i^v \), the travel time for vehicle \( v \) from node \( i \) to node \( j \) is \( t_{ij}^v \), and finally \( x_{ij}^v = 1 \) if arc \( i-j \) is used by vehicle \( v \). From this, the formulation of the GVRP is as follows:

\[
\text{Minimize } Z(t) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{v=1}^{w} c_{ij} x_{ij}^v \\
(A3.1)
\]

Subject to:

\[
\sum_{i=1}^{n} \sum_{v=1}^{w} x_{ij}^v = 1 \quad (j = 2, \ldots, n) \\
(A3.2)
\]

\[
\sum_{j=1}^{n} \sum_{v=1}^{w} x_{ij}^v = 1 \quad (i = 2, \ldots, n) \\
(A3.3)
\]

\[
\sum_{i=1}^{n} x_{ip}^v - \sum_{j=1}^{n} x_{pj}^v = 0 \quad (v = 1, \ldots, w; p = 1, \ldots, n) \\
(A3.4)
\]

\[
\sum_{i=1}^{n} d_i (\sum_{j=1}^{n} x_{ij}^0) \leq K_v \quad (v = 1, \ldots, w) \\
(A3.5)
\]

53
\[
\sum_{j=1}^{n} s_{ij} \sum_{j=1}^{n} x_{ij}^v + \sum_{j=1}^{n} t_{ij}^v x_{ij}^v \leq T_v \quad (v = 1, \ldots, w) \tag{A3.6}
\]

\[
\sum_{j=2}^{n} x_{ij}^v \leq 1 \quad (v = 1, \ldots, w) \tag{A3.7}
\]

\[
\sum_{i=2}^{n} x_{ii}^v \leq 1 \quad (v = 1, \ldots, w) \tag{A3.8}
\]

where \( X = (x_{ij}^v) \in S \), \( x_{ij}^v \in \{0,1\} \) \( \forall \ i, j, v \).  

The objective function, which minimizes the overall distance, remains the same but is formulated to sum over all vehicles. Equations (A3.2) and (A3.3) require that every customer is visited by exactly one vehicle. It is assumed that a customer’s demand does not exceed vehicle capacity and that each customer is fully serviced by the single vehicle that visits it. Equation (A3.4) requires continuity of our routes while (A3.5) maintains the vehicle capacity constraint. Since route length restrictions are represented with times, equation (A3.6) requires that maximum route duration is not exceeded. Finally, equations (A3.7) and (A3.8) limit the number of vehicles used.

In addition to these equations, subtour breaking constraints, slightly modified from those used earlier in the TSP, must be included. Since it is the most efficient, the third subtour representation is selected for expansion as follows:

\[
S = \left\{ x_{ij}^v : y_i^v - y_j^v + n x_{ij}^v \leq n - 1 \text{ for } 2 \leq i \neq j \leq n \text{ for some real numbers } y_i^v \right\}
\]

This applies the original subtour breaking constraint to each vehicle in turn. We note that some redundant constraints can be eliminated from the formulation above.

Using (A3.2) and (A3.4) enforces (A3.3) automatically and makes it unnecessary (Bodin
et al. 1983). Likewise (A3.4) and (A3.7) imply (A3.8) so this too can be eliminated from
the formulation (Bodin et al. 1983).

Finally, one common constraint added to the VRP is time windows. Let \( a_j \) be the
arrival time to node \( j \), \( e_j \) be the earliest delivery time allowable and \( l_j \) be the no later than
time for delivery. A nonlinear representation yields

\[
a_j = \sum_v \sum_i (a_i + s_i^v + t_i^v) x_{ij}^v \quad (j = 1, 2, \ldots, n)
\]

\[
a_i = 0
\]

\[
e_j \leq a_j \leq l_j \quad (j = 2, \ldots, n)
\]

If \( x_{ij}^v = 0 \) then \( a_j = 0 \). Otherwise \( a_j \) is the sum of the previous arrival time \( (a_i = 0) \), the
service time at node \( i \) (\( s_i^v \)), and the travel time from \( i \) to \( j \) (\( t_i^v \)). Alternatively the linear
representation of time windows constraint (Bodin et al. 1983) can be used in the
formulation

\[
a_j \geq (a_i + s_i^v + t_i^v) - (1 - x_{ij}^v) \cdot T_{\text{max}}^v
\]

\[
a_j \leq (a_i + s_i^v + t_i^v) + (1 - x_{ij}^v) \cdot T_{\text{max}}^v
\]

for all \( i, j, v \). (A3.12)

When \( x_{ij}^v = 1 \), the second half of the equation is eliminated and \( a_j \) is determined
from the previous arrival time, previous service time, and the travel time between the
nodes. When \( x_{ij}^v = 0 \), the constraints are redundant.

### A.4 Multiple Depot Vehicle Routing Problem (MDVRP)

Expanding the previous GVRP to account for multiple depots, or bases of
operation, gives the multiple depot VRP. This problem can be formulated with only
minor changes. Let $M$ be the number of depots in our problem. First the original VRP formulation indexes are changed for equation (A3.2), ($j = M + 1, \ldots, n$), and equation (A3.3), ($i = M + 1, \ldots, n$). Next the constraints (A3.7) and (A3.8) are changed to sum over all the depots individually to require that the number of vehicles used does not exceed the number of vehicles available.

$$\sum_{i=1}^{M} \sum_{j=M+1}^{n} x_{ij}^v \leq 1 \quad (v = 1, \ldots, w)$$

$$\sum_{p=1}^{M} \sum_{i=M+1}^{n} x_{ip}^v \leq 1 \quad (v = 1, \ldots, w)$$

The MDVRP also requires an adjustment to the subtour breaking constraint.

Again, only one is required (Bodin et al. 1983).

1. $S = \{(x_{ij}): \sum_{i \in Q} \sum_{j \in Q} x_{ij} \geq 1 \text{ for every nonempty proper subset } Q \text{ of } \{1, 2, \ldots, n\}\}$;

2. $S = \{(x_{ij}): \sum_{i \in R} \sum_{j \in R} x_{ij} \leq |R| - 1 \text{ for every nonempty subset } R \text{ of } \{M + 1, M+2, \ldots, n\}\}$;

3. $S = \{(x_{ij}): y_i - y_j + nx_{ij} \leq n - 1 \text{ for } M + 1 \leq i \neq j \leq n \text{ for some real numbers } y_i\}$.
Appendix B. Tabu Search vs. Other Heuristics—TSP Example

Objective: Minimize distance, $d$

Initial Order, $d = 3138$

Global Greedy, $d = 2238$

Nearest Neighbor, $d = 2108$

Tabu Search, $d = 1830$

*Nari* Data Set (Sisson 1997)
Appendix C. Javadoc Listing

Class Hierarchy
- class java.lang.Object
- class Convert
- class CoordType
- class CycleOut
- class HashMod
- class InFromKeybd
- class KeyObj
- class KeyToString
- class MTSPTWay
- class BestSolnMod
- class TsprwPen
- class NoCycleOut
- class NodeType
- class PrintCalls
- class PrintFlag
- class ReactTabuObj
- class ReadFile
- class SearchOut
- class StartPenBestOut
- class StartTourObj
- class TabuMod
- class TimeMatrixObj
- class Timer
- class TspTwPenOut
- class TwBestTTOut
- class ValueObj
- class VrpPenType
- class WindAdjust
- class WindData

Index of all Fields and Methods

A

acspeed. Variable in class WindData
double aircraft speed at the associated altitude level.

altitude. Variable in class WindData
integer value of the associated altitude level.

assignInputFile(String). Static method in class ReadFile
assignInputFile sets up the FileInputStream.

B

bands. Variable in class WindData
double Number of altitude level bands.
**bearing**. Variable in class `WindData`
double wind bearing at the associated altitude level.

**bearingXY** (CoordType, CoordType, double). Static method in class `Convert`
bearingXY calculates the true bearing (in degrees) from one coordinate point to the second
coordinate point and returns the value as a double precision number.

**bestCost**. Variable in class `SearchOut`

**bestCost**. Variable in class `StartPenBestOut`
Penalty related value.

**bestCost**. Variable in class `TwBestTTOut`
best tour related value.

**bestTiter**. Variable in class `SearchOut`

**bestTiter**. Variable in class `StartPenBestOut`
Penalty related value.

**bestTiter**. Variable in class `TwBestTTOut`
best tour related value.

**bestTny**. Variable in class `SearchOut`

**bestTny**. Variable in class `StartPenBestOut`
Penalty related value.

**bestTny**. Variable in class `TwBestTTOut`
best tour related value.

**BestSolnMod**(). Constructor for class `BestSolnMod`

**bestTime**. Variable in class `SearchOut`

**bestTime**. Variable in class `StartPenBestOut`
Penalty related value.

**bestTime**. Variable in class `TwBestTTOut`
best tour related value.

**bestTour**. Variable in class `SearchOut`

**bestTour**. Variable in class `StartPenBestOut`
Saved tour.

**bestTour**. Variable in class `TwBestTTOut`
best tour related value.

**bestTT**. Variable in class `SearchOut`

**bestTT**. Variable in class `StartPenBestOut`
Penalty related value.

**bestTT**. Variable in class `TwBestTTOut`
best tour related value.

**bfCost**. Variable in class `SearchOut`

**bfCost**. Variable in class `StartPenBestOut`
Penalty related value.

**bfCost**. Variable in class `TwBestTTOut`
best tour related value.

**bfTiter**. Variable in class `SearchOut`

**bfTiter**. Variable in class `StartPenBestOut`
Penalty related value.

**bfTiter**. Variable in class `TwBestTTOut`
best tour related value.

**bfTny**. Variable in class `SearchOut`

**bfTny**. Variable in class `StartPenBestOut`
Penalty related value.

**bfTny**. Variable in class `TwBestTTOut`
best tour related value.

**bfTime**. Variable in class `SearchOut`

**bfTime**. Variable in class `StartPenBestOut`
Penalty related value.
bfTime. Variable in class TwBestTTOut
best tour related value.

bfTour. Variable in class SearchOut
bfTour. Variable in class StartPenBestOut
Saved tour.

bfTour. Variable in class TwBestTTOut
best tour related value.

bfTT. Variable in class SearchOut
bfTT. Variable in class StartPenBestOut
Penalty related value.

bfTT. Variable in class TwBestTTOut
best tour related value.

C

compPems(NodeType[], int). Static method in class NodeType
compPems computes the vehicle capacity overload and time window penalties.

compPens(NodeType[], int). Method in class VrpPenType
compPens computes the vehicle capacity overload and time window penalties.

Convert(). Constructor for class Convert

CoordType(). Constructor for class CoordType
Default constructor.

CoordType(String, double, double). Constructor for class CoordType
Lat/long constructor.

copy(). Method in class NodeType

countVeh(NodeType[]). Static method in class NodeType
Method countVeh finds the number of vehicles being used in the current tour by counting the
vehicle to demand transitions.

countVehicles(NodeType[]). Static method in class TabuMod
countVeh method calculates the number of vehicles used in the current tour by counting the
number of vehicle (type 2) to demand (type 1) transitions.

cycle(ValueObj, double, int, int, int, double, int, PrintFlag). Static method in class TabuMod
cycle method updates the search parameters if the incumbent tour is found in the hashing structure.

CycleOut(). Constructor for class CycleOut
Default constructor.

CycleOut(int, double, ValueObj). Constructor for class CycleOut
Specified constructor.

cyclePrint. Variable in class PrintFlag
print flag.

D

distanceXY(CoordType, CoordType). Static method in class Convert
distanceXY calculates the great circle distance (in nautical miles) between two coordinate points
and returns the value as a double precision number.

DMMntoDd(int, double). Static method in class Convert
DMMntoDd converts a number in "Degrees Minutes Decimal Minutes" (D.MMm) format to
"Degrees Decimal Degrees" (D.d) format.

DMMntoDd(int, double, String). Static method in class Convert
dDMMntoDd converts a number in "Degrees Minutes Decimal Minutes" (D.MMm) format to
"Degrees Decimal Degrees" (D.d) format.

DMSSstoDd(int, int, double). Static method in class Convert
DMSSstoDd converts a number in "Degrees Minutes Seconds Decimal Seconds" (D.MMSSs) format to "Degrees Decimal Degrees" (D.d) format.

DMSSstoDd(int, int, double, String). Static method in class Convert
dDMSSstoDd converts a number in "Degrees Minutes Seconds Decimal Seconds" (D.MMSSs) format to "Degrees Decimal Degrees" (D.d) format.
E

endTime. Variable in class Timer
endTime time.
endTime(). Method in class Timer
endTime assigns endTime.
equals(KeyObj). Method in class KeyObj
Overloaded equals(), check only attribute fields.
equals(ValueObj). Method in class ValueObj
Overloaded equals(), check only attribute fields.

F

firstHashVal(int). Static method in class HashMod
firstHashVal method assigns the primary hashing value.

G

getACspeed(int). Method in class WindData
getACspeed returns aircraft (UAV) speed for the specified band.
getAltitude(int). Method in class WindData
getAltitude returns actual altitude for the specified band.
getBands(). Method in class WindData
getBands returns number of altitude bands (wind tiers).
getBearing(int). Method in class WindData
getBearing returns wind bearing for the specified band.
getEa(). Method in class NodeType
gerLoad(). Method in class NodeType
gerL(). Method in class NodeType
getLoad(). Method in class NodeType
gerM(). Method in class NodeType
getQty(). Method in class NodeType
getSpeed(int). Method in class WindData
getSpeed returns wind speed for the specified band.
getType(). Method in class NodeType
gerWait(). Method in class NodeType
groundSpeed(double, double, double, double). Static method in class WindAdjust
groundSpeed method returns the ground speed given the heading between points, the wind
toSpeed, the wind speed, and the aircraft's airspeed.
groundSpeedAF(double, double, double, double). Static method in class WindAdjust
groundSpeedAF is an experimental method that uses a different formula.

H

hashCode(). Method in class KeyObj
hashCode() method.
hashCode(). Method in class ValueObj
hashCode() method.
HashMod(). Constructor for class HashMod
HHMMtoMM(int). Static method in class Convert
HHMMtoMM converts a military time to the equivalent number of minutes (i.e., 0630 hours to
390 minutes) for use in time window and service time calculations.
HHMtoHh(int). Static method in class Convert
HHMtoHh converts a military specified time to the equivalent decimal hour equivalent (i.e., 0630
hours to 6.5 hours) for use in time window and service time calculations.

I

InFromKeybd(). Constructor for class InFromKeybd
**insert(NodeType[], int, int).** Static method in class **NodeType**
Method insert allows the element designated by "chI" to be shifted by "chD" elements.

**iterPrint.** Variable in class **PrintFlag**
print flag.

**K**

**keyDouble(String).** Static method in class **InFromKeybd**
keyDouble allows user to enter a double from the keyboard.

**keyFloat(String).** Static method in class **InFromKeybd**
keyFloat allows user to enter a float from the keyboard.

**keyInt(String).** Static method in class **InFromKeybd**
keyInt allows user to enter an integer from the keyboard.

**KeyObj(int, int, int, int, int).** Constructor for class **KeyObj**
Specified constructor.

**keyString(String).** Static method in class **InFromKeybd**
keyString allows user to enter a string from the keyboard.

**KeyToString().** Constructor for class **KeyToString**

**KeyToString(int, int, int, int, int, int).** Static method in class **KeyToString**
KeyToString Class converts the attributes of tour to a concatenated string used as a key to the hashtable of tours.

**L**

**loadPrint.** Variable in class **PrintFlag**
print flag.

**lookFor(HashMap, int, int, int, int, int, int, int).** Static method in class **HashMod**
lookFor method searches for the current tour in the hashing structure, if the tour is found a true value for the boolean "found" is returned, if not found, the tour is added to the hashtable.

**M**

**main(String[]).** Static method in class **MTSPTWuav**
main executes MTSPTWuav problem.

**mavg.** Variable in class **CycleOut**
moving average.

**MMtoHMM(int).** Static method in class **Convert**
MMtoHMM converts a given number of minutes to a military time hour format (i.e., 390 minutes to 0630 hours) for human friendly output.

**movePrint.** Variable in class **PrintFlag**
print flag.

**moveValTT(int, int, NodeType[], NodeType[], int[][]).** Static method in class **NodeType**
Method moveValTT computes the incremental change in the value of the travel time from the incumbent tour to the proposed neighbor tour, and computes the neighbor schedule parameters preparing for computation of penalty terms.

**moveValTT(int, int, NodeType[], NodeType[], int[][]).** Static method in class **TabuMod**
Method moveValTT computes the incremental change in the value of the travel time from the incumbent tour to the proposed neighbor tour, and computes the neighbor schedule parameters preparing for computation of penalty terms.

**MTSPTWuav().** Constructor for class **MTSPTWuav**

**N**

**noCycle(double, int, double, int, int, PrintFlag).** Static method in class **TabuMod**
noCycle method updates the search parameters if the incumbent tour is not found in the hashing structure.

**NoCycleOut().** Constructor for class **NoCycleOut**
Default constructor.

**NoCycleOut(int, int).** Constructor for class **NoCycleOut**
Specified constructor.
NodeType(). Constructor for class NodeType
   Default constructor.

NodeType(int, int, int, int, int, int). Constructor for class NodeType
   Specified constructor.

numfeas. Variable in class SearchOut

p

penTray. Variable in class SearchOut

penTray. Variable in class StartPenBestOut
   Penalty related value.

penTray. Variable in class TsptwPenOut
   Penalty related value.

print(). Method in class NodeType

PrintCalls(). Constructor for class PrintCalls

PrintFlag(). Constructor for class PrintFlag
   Default PrintFlag constructor sets all to "true".

PrintFlag(boolean). Constructor for class PrintFlag
   Additional PrintFlag constructor allows specification of either "true" or "false".

printInitVals(int, int, int, double, String). Static method in class PrintCalls

printTour(NodeType[]). Static method in class NodeType

R

randWtWZ(int, int, int). Static method in class HashMod
   randWtWZ method computes Woodruff & Zemel random weights between 1 & range for all nodes.

ReacTabuObj(). Constructor for class ReacTabuObj

ReadFile(). Constructor for class ReadFile

readNC(String). Static method in class TimeMatrixObj
   readNC is used to read from the first token from the input file (the number of customers (nc)).

readNextDouble(Tokenizer). Static method in class ReadFile
   readNextString method gets the next token and returns it as a double.

readNextInt(Tokenizer). Static method in class ReadFile
   readNextString method gets the next token and returns it as an integer.

readNextString(Tokenizer). Static method in class ReadFile
   readNextString method gets the next token and returns it as a string.

readNV(String). Static method in class TimeMatrixObj
   readNV is used to read from the second token from the input file (the number of vehicles (nv)).

readTSPTW(double, int, String, CoordType[], int[]). Static method in class TimeMatrixObj
   readTSPTW reads in the geographical coordinates and time window file and calculates the time
time between each node.

readTSPTWdepot(double, int, int, String, CoordType[], int[]). Static method in class TimeMatrixObj
   readTSPTWdepot reads in the geographical coordinates, load quantity, service time, and time
window information associated with depot and customer locations from the input file.

readTSPTWdepotUAV(double, int, int, String, CoordType[], int[]). Static method in class
   TimeMatrixObj
   readTSPTWdepotUAV reads in the geographical coordinates, load quantity, min/max service
times, and time window information associated with depot and target locations from the input file.

readTSPTWrwrouteUAV(double, int, int, String, CoordType[], int[]). Static method in class
   TimeMatrixObj
   readTSPTWrwrouteUAV reads in the geographical coordinates, load quantity, service time, and
time window information associated with depot and customer locations from the input file.

readWind(String). Static method in class WindData
   readWind method reads wind data from a file and returns a WindData object.

rtsStepPrint(int, int, int, int, int, int). Static method in class PrintCalls
search(double, double, double, double, int, int, int, int, int, int, int, int, int, int, VrpPenType, int[][], PrintFlag, int, int, int, int, int, int, int, int, int, NodeExt[], NodeType[], NodeType[]). Static method in class ReactTabuObj
  ReactTabuObj steps through iterations of the reactive tabu search.

SearchOut(). Constructor for class SearchOut
  Default constructor.

SearchOut(int, int, int, int, int, int, int, int, int, int, int, int, int, int, VrpPenType, NodeType[], NodeType[], NodeType[]). Constructor for class SearchOut
  Specified constructor.

secondHashVal(int, int, NodeType[], int[]). Static method in class HashMod
  secondHashVal updates the Woodruff & Zemel second hashing value based on the tour insertion move.

setACspeed(int, int). Method in class WindData
  setACspeed sets only the UAV speed information only for a particular band.

setACspeedAll(int). Method in class WindData
  setACspeedAll sets only the UAV speed information.

setAll(int, int, double, double, double, int). Method in class WindData
  setAll method sets all related information for a particular band.

setAlt(int, int). Method in class WindData
  setAltitude sets only the altitude information for a particular band.

setBearing(int, double). Method in class WindData
  setBearing sets only the bearing information only for a particular band.

setId(int). Method in class NodeType

setLoad(int). Method in class NodeType

setQty(int). Method in class NodeType

setSpeed(int, double). Method in class WindData
  setSpeed sets only the wind speed information only for a particular band.

setType(int). Method in class NodeType

setWait(int). Method in class NodeType

setWind(int, int, double, double). Method in class WindData
  setWind method sets only wind related information for a particular band.

speed. Variable in class WindData
  double wind speed at the associated altitude level.

ssltc. Variable in class CycleOut

ssltc. Variable in class NoCycleOut
  cycle related variable.

startPenBest(int, int, NodeType[], double, double, int, int, int, int, VrpPenType, int, int, int, int, int, NodeExt[], NodeType[], NodeType[]). Static method in class StartTourObj
  startPenBest initializes "best" values and their times.

StartPenBestOut(). Constructor for class StartPenBestOut
  Default constructor.

StartPenBestOut(int, int, int, int, int, int, int, int, int, int, int, int, int, int, VrpPenType, NodeType[], NodeType[]). Constructor for class StartPenBestOut
  Specified constructor.

startPrint. Variable in class PrintFlag
  print flag.

startTime. Variable in class Timer
  begin time.

startTime(). Method in class Timer
  startTime assigns start time.

StartTour(NodeExt[], int[], int, int). Static method in class NodeType
  Method startTour will bubble sort the initial tour based on the average time window time.

StartTourObj(). Constructor for class StartTourObj
**stepLoopPrint**: Variable in class PrintFlag
   print flag.

**stepPrint**: Variable in class PrintFlag
   print flag.

**sumWait(NnodeType[]):** Static method in class Node
   Method sumWait calculates the total "waiting" time in a particular tour by summing the wait values for each individual node.

**swap(int, int):** Static method in class MTSPTWuav
   Swap allows generic swap of integers.

**swapInt(int, int):** Static method in class Node
   Method swapInt switches two integers

**swapNode(NnodeType[], int, int):** Static method in class Node
   Method swapNode allows the node array elements "a" and "b" to be swapped in the Node Array "z".

**T**

**tabuLen**: Variable in class CycleOut
   cycle related variable.

**tabuLen**: Variable in class NoCycleOut
   cycle related variable.

**TabuMod()**: Constructor for class TabuMod

**timeMatrix(int, int, double, int, CoordType[], int[]):** Static method in class TimeMatrixObj
   timeMatrix computes simple two-dimensional time/distance matrix.

**timeMatrixDepot(int, int, double, int, CoordType[], int[]):** Static method in class TimeMatrixObj
   timeMatrixDepot computes the two-dimensional array used as the "time" matrix.

**timeMatrixDepotUAV(int, int, double, int, CoordType[], int[]):** Static method in class TimeMatrixObj
   timeMatrixDepotUAV computes the two-dimensional array used as the "time" matrix.

**TimeMatrixObj()**: Constructor for class TimeMatrixObj

**timeMatrixUAV(int, int, double, int, CoordType[], int[], WindData, int[]):** Static method in class TimeMatrixObj
   timeMatrixDepot computes the two-dimensional array used as the "time" matrix.

**timeMatrixUAVroute(int, int, double, int, CoordType[], int[], WindData, int[]):** Static method in class TimeMatrixObj
   timeMatrixDepotUAVroute computes the two-dimensional array used as the "time" matrix.

**timePrint**: Variable in class PrintFlag
   print flag.

**Timer()**: Constructor for class Timer
   Default constructor.

**toString()**: Method in class KeyObj
   toString changes a KeyObj to a string for use in the hashTable.

**toString()**: Method in class ValueObj
   toString changes a ValueObj to a string for use in the hashTable.

**totalSeconds**: Variable in class Timer
   duration of run.

**totalSeconds()**: Method in class Timer
   totalSeconds returns duration.

**totPenalty**: Variable in class SearchOut

**totPenalty**: Variable in class StartPenBestOut
   Penalty related value.

**totPenalty**: Variable in class TsptwPenOut
   Penalty related value.

**tour**: Variable in class SearchOut

**tourCost**: Variable in class SearchOut

**tourCost**: Variable in class StartPenBestOut
   Penalty related value.
tourCost. Variable in class TsptwPenOut
Penalty related value.
tourHVwz(NodeType[], int[]). Static method in class HashMod
tourHVwz method computes the Woodruff & Zemel hashing value from the sum of adjacent node id multiplication.
tourPen. Variable in class SearchOut
tourPen. Variable in class StartPenBestOut
Tour penalty values.
tourSched(int, NodeType[], int[][]). Static method in class NodeType
Method tourSched should be called with the syntax tourLen = tourSched(nodeArray, time) from the orderStartingTour method.
TsptwPen(). Constructor for class TsptwPen
TsptwPen(int, NodeType[], VrpPenType, double, double, int, int, int, int). Static method in class TsptwPen
TsptwPen method uses the TW and load penalties to computes tourCost of tour as tour length + scaled penalty for infeasibilities.
TsptwPenNormalized(int, NodeType[], VrpPenType, double, double, int, int, int, int). Static method in class TsptwPen
TsptwPenNormalized method uses the TW and load penalties to computes tourCost of tour as tour length + scaled penalty for infeasibilities.
TsptwPenOut(). Constructor for class TsptwPenOut
Default constructor.
TsptwPenOut(int, int, int, int). Constructor for class TsptwPenOut
Specified constructor.
tvl. Variable in class SearchOut
tvl. Variable in class TsptwPenOut
Penalty related value.
twBestTT(int, int, int, int, int, NodeType[], int, int, int, int, NodeType[], NodeType[],
int, int). Static method in class BestSolnMod
twBestTT compares current tour with previous best and best feasible tours and updates records accordingly.
TwBestTTOut(). Constructor for class TwBestTTOut
Default constructor.
TwBestTTOut(int, int, int, int, int, int, int, int, NodeType[], NodeType[]). Constructor for class TwBestTTOut
Specified constructor.
twrdPrint. Variable in class PrintFlag
print flag.
V
ValueObj(int, int, int, int, int, int). Constructor for class ValueObj
Specified constructor.
VrpPenType(). Constructor for class VrpPenType
Default constructor.
VrpPenType(int, int). Constructor for class VrpPenType
Specified constructor.
VrpPenType(int, int). Constructor for class VrpPenType
Specified constructor.
W
WindAdjust(). Constructor for class WindAdjust
**Class BestSolnMod**

```
java.lang.Object
   +---MTSPTWuav
   |    +---BestSolnMod
```

public class BestSolnMod extends MTSPTWuav
BestSolnMod class retains the tours with the best travel times and tour costs.

Version:
   v1.1 Mar 99
Author:
   Kevin P. O'Rourke, David M. Ryer

**Constructor Index**

**BestSolnMod()**

**Method Index**

**twBestTT(int, int, int, int, int, int, NodeTye[], int, int, int, int, int, int, int, int, NodeType[], NodeType[],
   int, int)**

twBestTT compares current tour with previous best and best feasible tours and updates records accordingly.

**Constructors**

BestSolnMod

```
public BestSolnMod()
```

**Methods**

**twBestTT**

```
public static TwBestTTOut twBestTT(int numnodes,
   int totPenalty,
   int penTrav,
   int tvl,
   int nvu,
   int iter,
   NodeType tour[],
   int bfCost,
   int bTT,
   int bfpr,
   int bfiter,
   int bestCost,
   int bestTT,
   int bestnv,
   int bestiter,
   NodeType bfTour[],
   NodeType bestTour[],
   int bftime,
```

67
int bestTime

twBestTT compares current tour with previous best and best feasible tours and updates records accordingly.

Returns:
returns packages output object.

Class Convert

java.lang.Object
    |
    +----+Convert

public class Convert
extends Object Convert contains general conversion formulas applicable to location and distance calculations. Included are conversions between decimal format and hours-minutes-seconds format, great circle distance between two specified coordinates, and bearing from one point to another.

Version:
v1.1 Feb 99

Author:
Kevin P. O'Rourke, David M. Ryer

Constructor Index

Constructor

Method Index

bearingXY(CoordType, CoordType, double)
    bearingXY calculates the true bearing (in degrees) from one coordinate point to the second coordinate point and returns the value as a double precision number.

distanceXY(CoordType, CoordType)
    distanceXY calculates the great circle distance (in nautical miles) between two coordinate points and returns the value as a double precision number.

DMMtoDD(int, double)
    DMMtoDD converts a number in “Degrees Minutes Decimal Minutes” (D.MMm) format to “Degrees Decimal Degrees” (D.d) format.

DMMtoDD(int, double, String)
    DMMtoDD converts a number in “Degrees Minutes Decimal Minutes” (D.MMm) format to “Degrees Decimal Degrees” (D.d) format.

DMMSSstoDD(int, int, double)
    DMMSSstoDD converts a number in “Degrees Minutes Seconds Decimal Seconds” (D.MMSSs) format to “Degrees Decimal Degrees” (D.d) format.

DMMSSstoDD(int, int, double, String)
    DMMSSstoDD converts a number in “Degrees Minutes Seconds Decimal Seconds” (D.MMSSs) format to “Degrees Decimal Degrees” (D.d) format.

HHMMtoMM(int)
    HHMMtoMM converts a military time to the equivalent number of minutes (i.e., 0630 hours to 390 minutes) for use in time window and service time calculations.

HHMtoHh(int)
    HMMtoHh converts a military specified time to the equivalent decimal hour equivalent (i.e., 0630 hours to 6.5 hours) for use in time window and service time calculations.

MintoHHMM(int)
    MintoHHMM converts a given number of minutes to a military time hour format (i.e., 390 minutes to 0630 hours) for human friendly output.
Constructors
Convert

    public Convert()

Methods
DMMmtoDd

    public static double DMMmtoDd(int degrees,
         double minutes)

    DMMmtoDd converts a number in "Degrees Minutes Decimal Minutes" (D.MMm) format to
    "Degrees Decimal Degrees" (D.d) format. The D.MMm is the "human friendly" form of the data.
    The D.d format is required to readily perform distance calculations.

Parameters:
    degrees - integer degree value of coordinate.
    minutes - double minute value of coordinate.

Returns:
    returns double Dd coordinate in the "degrees decimal degrees" format.

DMMMtoDd

    public static double DMMmtoDd(int degrees,
         double minutes,
         String name)

    DMMmtoDd converts a number in "Degrees Minutes Decimal Minutes" (D.MMm) format to
    "Degrees Decimal Degrees" (D.d) format. The D.MMm is the "human friendly" form of the data.
    The D.d format is required to readily perform distance calculations. This version of the method
    considers hemisphere and assigns a negative value if appropriate to south and east coordinates.

Parameters:
    degrees - integer degree value of coordinate.
    minutes - double minute value of coordinate.
    name - string hemisphere value of coordinate (either "E", "W", "N", or "S").

Returns:
    returns Dd coordinate in the "degrees decimal degrees" format.

DMMSsstoDd

    public static double DMMSsstoDd(int degrees,
         int minutes,
         double seconds)

    DMMSsstoDd converts a number in "Degrees Minutes Seconds Decimal Seconds" (D.MMSSs)
    format to "Degrees Decimal Degrees" (D.d) format. The D.MMSSs is the "human friendly" form
    of the data. The D.d format is required to readily perform distance calculations.

Parameters:
    degrees - integer degree value of coordinate.
    minutes - integer minute value of coordinate.
    seconds - double second value of coordinate.

Returns:
    returns Dd coordinate in the "degrees decimal degrees" format.

DMMSsstoDd

    public static double DMMSsstoDd(int degrees,
         int minutes,
         double seconds,
         String name)

    DMMSsstoDd converts a number in "Degrees Minutes Seconds Decimal Seconds" (D.MMSSs)

69
format to "Degrees Decimal Degrees" (D.d) format. The D.MMSSs is the "human friendly" form of the data. The D.d format is required to readily perform distance calculations. This version of the method considers hemisphere and assigns a negative value if appropriate to south and east coordinates.

Parameters:
- degrees - integer degree value of coordinate.
- minutes - integer minute value of coordinate.
- seconds - double second value of coordinate.
- name - string hemisphere value of coordinate (either "E", "W", "N", or "S").

Returns:
- returns Dd coordinate in the "degrees decimal degrees" format.

HMMtoHh

public static double HMMtoHh(int time)
    HMMtoHh converts a military specified time to the equivalent decimal hour equivalent (i.e., 0630 hours to 6.5 hours) for use in time window and service time calculations.

Parameters:
- time - integer whole minute "military format" (0630 hours) time value.

Returns:
- returns Hh double fractional hour (6.5 hours) time value.

HHMtoMM

public static int HHMtoMM(int time)
    HHMtoMM converts a military time to the equivalent number of minutes (i.e., 0630 hours to 390 minutes) for use in time window and service time calculations.

Parameters:
- time - integer whole minute "military format" (0630 hours) time value.

Returns:
- returns MM integer number of minutes (390 minutes) time value.

MMtoHHMM

public static int MMtoHHMM(int time)
    MMtoHHMM converts a given number of minutes to a military time hour format (i.e., 390 minutes to 0630 hours) for human friendly output.

Parameters:
- time - integer number of minutes (390 minutes) time value.

Returns:
- returns HHMM integer whole minute "military format" (0630 hours) time value.

distanceXY

public static double distanceXY(CoordType x, CoordType y)
    distanceXY calculates the great circle distance (in nautical miles) between two coordinate points and returns the value as a double precision number.

Parameters:
- x - CoordType coordinate of first position.
- y - CoordType coordinate of second position.

Returns:
- returns distanceXY double distance between the two points in nautical miles.

bearingXY

public static double bearingXY(CoordType x, CoordType y, double dXY)

70
bearingXY calculates the true bearing (in degrees) from one coordinate point to the second coordinate point and returns the value as a double precision number.

Parameters:
- x - CoordType coordinate of first position.
- y - CoordType coordinate of second position.
- dXY - double distance between the first and second position, in nautical miles.

Returns:
- returns thetaXY double initial true heading from the first point to the second point measured from true north in degrees.

**Class CoordType**

```
java.lang.Object
    +---- CoordType
```

public class CoordType extends Object CoordType is used to hold coordinate location for customer/vehicle nodes. It contains fields for both x, y integer data and lat/long data, although only one set will be used.

Version:
- v1.1 Feb 99

Author:
- Kevin P. O'Rourke, David M. Ryer

**Constructor Index**

```
CoordType()
    Default constructor.
CoordType(String, double, double)
    Lat/long constructor.
```

**Constructors**

```
CoordType
    public CoordType()
        Default constructor. Assigns name to null and all values to zero.
CoordType
    public CoordType(String nameLabel,
                      double lat,
                      double lon)
        Lat/long constructor. Assigns name, latitude, and longitude as specified.
```

**Class CycleOut**

```
java.lang.Object
    +---- CycleOut
```

public class CycleOut extends Object CycleOut is used as a package to output multiple fields from the class Cycle.
Version:
   v1.1 Mar 99
Author:
   Kevin P. O'Rourke, David M. Ryer

*Variable Index*

**mavg**
   moving average.

**ssl1c**

**tabuLen**

*Constructor Index*

**CycleOut()**
   Default constructor.

**CycleOut(int, int, double, ValueObj)**
   Specified constructor.

**ssl1c**

    public int ssl1c

**tabuLen**

    public int tabuLen

**mavg**

    public double mavg
       moving average.

*Constructors*

**CycleOut**

    public CycleOut()
       Default constructor. Assigns all values to zero.

**CycleOut**

    public CycleOut(int ssl1c,
                    int tabuLen,
                    double mavg,
                    ValueObj matchPtr)
       Specified constructor. Values set as passed.

*Class HashMod*

    java.lang.Object
       |
       +---- HashMod

**public class HashMod**
exends Object HashMod Class contains methods to assign first and second hashing values (used in the hashtable) and the search method to search the hashtable.
Version:
v1.1 Mar 99

Author:
Kevin P. O'Rourke, David M. Ryer

Constructor Index

**HashMod()**

Method Index

`firstHashVal(int)`
firstHashVal method assigns the primary hashing value.

`lookFor(Hashtable, int, int, int, int, int, int)`
lookFor method searches for the current tour in the hashing structure, if the tour is found a true value for the boolean "found" is returned, if not found, the tour is added to the hashtable.

`randWtWZ(int, int, int)`
randWtWZ method computes Woodruff & Zemel random weights between 1 & range for all nodes.

`secondHashVal(int, int, int, NodeType[], int[])`
secondHashVal updates the Woodruff & Zemel second hashing value based on the tour insertion move.

`tourHVwz(NodeType[], int[])`
tourHVwz method computes the Woodruff & Zemel hashing value from the sum of adjacent node id multiplication.

Constructors

**HashMod**

`public HashMod()`

**Methods**

`lookFor`

`public static boolean lookFor(Hashtable daHashTab,`  
`int fhv,`  
`int shv,`  
`int cost,`  
`int tvl,`  
`int twPen,`  
`int loadPen,`  
`int lastIter)`

lookFor method searches for the current tour in the hashing structure, if the tour is found a true value for the boolean "found" is returned, if not found, the tour is added to the hashtable.

Parameters:
daHashTab - hashtable object.
fhv - First hashing value (objective function).
shv - Second hashing value (Woodruff & Zemel).
tourCost - Tour cost.
tvl - Travel time.
twPen - Time window penalty.
loadPen - Load overage penalty.
lastIter - Iteration on which the tour was previously found.

Returns:
returns true boolean value if the tour was previously found.

`randWtWZ`

`public static final int[] randWtWZ(int ZRANGE,`
int nc,
int numnodes)

randWtWZ method computes Woodruff & Zemel random weights between 1 & range for all
nodes.
Parameters:
    ZRANGE - maximum weight value.
    nc - number of customers (targets).
    numnodes - total number of nodes.
Returns:
    returns integer array of "z" weights.

tourHVwz

public static final int tourHVwz(NodeType tour[],
    int zArr[])

    tourHVwz method computes the Woodruff & Zemel hashing value from the sum of adjacent node
    id multiplication.
Parameters:
    tour - tour node array to be processed.
    zArr - "z" array of random weights.
Returns:
    returns secondary hashing value function (thv).

firstHashVal

public static final int firstHashVal(int zT)

    firstHashVal method assigns the primary hashing value. Currently, it assigns the objective
    function as the first hashing value (fhv). Method can be updated as desired.
Parameters:
    zT - objective function value.
Returns:
    returns first hashing value (fhv).

secondHashVal

public static final int secondHashVal(int shv,
    int chI,
    int chD,
    NodeType tour[],
    int zArr[])

    secondHashVal updates the Woodruff & Zemel second hashing value based on the tour insertion
    move.
Parameters:
    shv - current tour hashing value.
    chI - node insertion position.
    chD - node insertion depth.
    tour - tour node array for processing.
    zArr - "z" array of random weights.
Returns:
    returns updated hashing value to reflect insertion.

Class InFromKeybd

java.lang.Object
|
  +---- InFromKeybd
public class InFromKeybd
extends Object
InFromKeybd class allows us to enter strings, integers, doubles and floats from the
keyboard with a specified prompt.
Version:
   v1.1 Feb 99
Author:
   Kevin P. O'Rourke, David M. Ryer

Constructor Index
InFromKeybd()

Method Index
keyDouble(String)
   keyDouble allows user to enter a double from the keyboard.
keyFloat(String)
   keyFloat allows user to enter a float from the keyboard.
keyInt(String)
   keyInt allows user to enter an integer from the keyboard.
keyString(String)
   keyString allows user to enter a string from the keyboard.

Constructors
InFromKeybd
   public InFromKeybd()

Methods
keyString
   public static final String keyString(String prompt)
   keyString allows user to enter a string from the keyboard.
Parameters:
   prompt - Text prompt printed on screen.
Returns:
   returns user entered string.

keyInt
   public static final int keyInt(String prompt)
   keyInt allows user to enter an integer from the keyboard.
Parameters:
   prompt - Text prompt printed on screen.
Returns:
   returns user entered integer.

keyDouble
   public static final double keyDouble(String prompt)
   keyDouble allows user to enter a double from the keyboard.
Parameters:
   prompt - Text prompt printed on screen.
Returns:
   returns user entered double.

keyFloat
   public static final float keyFloat(String prompt)
   keyFloat allows user to enter a float from the keyboard.
Parameters:
  prompt - Text prompt printed on screen.
Returns:
  returns user entered float.

Class KeyObj

java.lang.Object
  +---KeyObj

public final class KeyObj
extends Object

KeyObj Class is used to access tour attributes in the hashtable for comparison.

Version:
  v1.1 Mar 99

Author:
  Kevin P. O'Rourke, David M. Ryer

Constructor Index

KeyObj(int, int, int, int, int, int)
  Specified constructor.

Method Index

equals(KeyObj)
  Overloaded equals(), check only attribute fields.
hashCode()
  Overloaded hashCode method.
toString()
  toString changes a KeyObj to a string for use in the hashTable.

Constructors

KeyObj

public KeyObj(int fhv,
  int shv,
  int cost,
  int tvl,
  int twPen,
  int loadPen)
  Specified constructor. Values set as passed.

Methods

equals

public final boolean equals(KeyObj a)
  Overloaded equals(), check only attribute fields. Do not check first two data elements to keep
  inline with hashCode overload.

Parameters:
  a - element compared calling object.
Returns:
  returns true if objects are equal, false otherwise.

toString

public final String toString()
toString changes a KeyObj to a string for use in the hashTable.

Returns:
returns concatenated String.

Overrides:
toString in class Object

hashCode

public final int hashCode()
        overloaded hashcode method. Note: if two objects are equal according to the equals method, then
calling the hashCode method on each of the two objects must produce the same integer result.
Only check first two data elements because of size limitations of Integer.

Returns:
returns integer hashcode value.

Override:
hashCode in class Object

Class KeyToString

java.lang.Object
  |
  +---- KeyToString

public class KeyToString
extends Object
KeyToString Class converts the attributes of tour to a concatenated string used as a key to
the hashtable of tours.
Version:
    1.1 Mar 99
Author:
    Kevin P. O'Rourke, David M. Ryer

Constructor Index

keyToString()

Method Index

keyToString(int, int, int, int, int)
    KeyToString Class converts the attributes of tour to a concatenated string used as a key to the
    hashtable of tours.

Constructors

KeyToString

    public KeyToString()

Methods

keyToString

    public static String keyToString(int fhv,
        int shv,
        int tourCost,
        int tvl,
        int twPen,
int loadPen)

KeyTostring class converts the attributes of tour to a concatenated string used as a key to the hash table of tours.

Parameters:
  fhv - First hashing value (objective function).
  shv - Second hashing value (Woodruff & Zemel).
  tourCost - Tour cost.
  tvl - Travel time.
  twPen - Time window penalty.
  loadPen - Load overage penalty.

Class MTSPTWuav

java.lang.Object
 |
+-----MTSPTWuav

Public class MTSPTWuav
extends Object MTSPTWuav is the main part that implements the multiple traveling salesperson problem with time windows solve algorithm. This version calls the UAV specific methods to read file input and generate the appropriate time matrix.

Version:
  v1.1 Mar 99

Author:
  Kevin P. O'Rourke, David M. Ryer

Constructor Index

MTSPTWuav()

Method Index

main(String[])  
  main executes MTSPTWuav problem.
swap(int, int)
  Swap allows generic swap of integers.

Constructors

MTSPTWuav

  public MTSPTWuav()

Methods

swap

  public static void swap(int a,
                           int b)
  Swap allows generic swap of integers.

Parameters:
  a - integer
  b - integer

Returns:
  returns void

main
public static void main(String argv[])  
    main executes MTSPTWuav problem. Initializes global variables, calls methods to read data and  
    wind files, calls method to compute time matrix, calls tabu search method, writes output to file.  
Parameters:  
    nv - number of vehicles, overridden by file information  
    iters - number of iterations  
    integer - precision scaling factor  
    file - data file name, without extension (actual filename must end with .dat).  
    wind - file name, without extension (actual filename must end with .dat).  
    reroute - identifier. Use 111 (one one one) to specify reroute.  

Class NoCycleOut  

java.lang.Object  
    |  
    +-----NoCycleOut  

public class NoCycleOut  
extends Object  
NoCycleOut is used as a package to output multiple fields from the method NoCycle.  
Version:  
v1.1 Mar 99  
Author:  
    Kevin P. O'Rourke, David M. Ryer  

Variable Index  

ssltc  
    cycle related variable.  

tabuLen  
    cycle related variable.  

Constructor Index  

NoCycleOut()  
    Default constructor.  
NoCycleOut(int, int)  
    Specified constructor.  

Variables  

ssltc  

public int ssltc  
    cycle related variable.  

tabuLen  

public int tabuLen  
    cycle related variable.  

Constructors  

NoCycleOut  

public NoCycleOut()  
    Default constructor. Assigns all values to zero.  
NoCycleOut
public NoCycleOut(int ss, int tabuLen)
    Specified constructor. Values set as passed.

Class Node

    java.lang.Object
    |
    +-----Node

public class Node extends Object Node defines the relevant information of each particular node.
Version:
    v1.1 Feb 99
Author:
    Kevin P. O'Rourke, David M. Ryer

Constructor Index

NodeType()
    Default constructor.
NodeType(int, int, int, int, int, int)
    Specified constructor.

Method Index

compPens(NodeType[], int)
    compPens computes the vehicle capacity overload and time window penalties.
copy()
countVeh(NodeType[])
    Method countVeh finds the number of vehicles being used in the current tour by counting the
    vehicle to demand transitions.
getEa()
getId()
getLa()
getLoad()
getM()
getQty()
getType()
getWait()
insert(NodeType[], int, int)
    Method insert allows the element designated by "chI" to be shifted by "chD" elements.
moveValTT(int, int, NodeType[], NodeType[], int[])[]
    Method moveValTT computes the incremental change in the value of the travel time from the
    incumbent tour to the proposed neighbor tour, and computes the neighbor schedule parameters
    preparing for computation of penalty terms.
print()
printTour(NodeType[])
setId(int)
setLoad(int)
setQty(int)
setType(int)
setWait(int)
startTour(NodeType[], int[], int, int)
    Method startTour will bubble sort the initial tour based on the average time window time.
sumWait(NodeType[])
Method sumWait calculates the total "waiting" time in a particular tour by summing the wait values for each individual node.

**swapInt**(int, int)
Method swapInt switches two integers

**swapNode**(NodeType[], int, int)
Method swapNode allows the node array elements "a" and "b" to be swapped in the Node Array "z".

**tourSched**(int, NodeType[], int[][])
Method tourSched should be called with the syntax tourLen = tourSched(nodeArray, time) from the orderStartingTour method.

**Constructors**

**NodeType**

```java
public NodeType()
    Default constructor. Assigns all values to zero.
```

**NodeType**

```java
public NodeType(int id,
    int ea,
    int la,
    int qty,
    int type,
    int wait,
    int load)
```

Specified constructor. Values set as passed.

**Methods**

**copy**

```java
public final NodeType copy()
```

**swapInt**

```java
public static final void swapInt(int a,
    int b)
```

Method swapInt switches two integers

**swapNode**

```java
public static final NodeType[] swapNode(NodeType z[],
    int a,
    int b)
```

Method swapNode allows the node array elements "a" and "b" to be swapped in the Node Array "z".

Parameters:
- z - node array to be updated.
- a - element to be swapped.
- b - element to be swapped.

Returns:
- returns updated node array.

**insert**

```java
public static final NodeType[] insert(NodeType z[],
    int chI,
    int chD)
```
Method insert allows the element designated by "chI" to be shifted by "chD" elements. chD may be positive or negative.

Parameters:
  z - node array to be updated.
  chI - location of node to be moved.
  chD - depth of move.

Returns:
  returns updated node array.

countVeh

public static final int countVeh(NodeType tour[])
Method countVeh finds the number of vehicles being used in the current tour by counting the vehicle to demand transitions.

Parameters:
  tour - node array to be processed.

Returns:
  returns integer number of vehicles used in the tour.

sumWait

public static final int sumWait(NodeType tour[])
Method sumWait calculates the total "waiting" time in a particular tour by summing the wait values for each individual node.

Parameters:
  tour - node array to be processed.

Returns:
  returns integer value of total wait time in the tour.

compPens

public static final VrpPenType compPens(NodeType tour[],
  int capacity)
compPens computes the vehicle capacity overload and time window penalties.

Parameters:
  tour[] - current tour used to calculate penalties.
  capacity - maximum vehicle load.

Returns:
  returns the VrpPenType object which the method was called on with updated values.

tourSched

public static final int tourSched(int is,
  NodeType tour[],
  int time[[][]])
Method tourSched should be called with the syntax tourLen = tourSched(nodeArray, time) from the orderStartingTour method. This will use the listing of nodes to return the new tourLen value (tour duration). Additionally, the nodeArray will be updated to reflect the new arrival and departure times.

Parameters:
  is - insertion/starting location for computation of schedule.
  tour - node array to be processed.
  time - time matrix used to determine schedule.

Returns:
  returns integer total tour duration. Updates tour node array as appropriate.

startTour

public static final int startTour(NodeType tour[]),
int time[][],
int nc,
int nv)

Method startTour will bubble sort the initial tour based on the average time window time. No
swap is made if the move would violate strong time window infeasibility.

Parameters:
tour - node array to be processed.
time - time matrix used to determine schedule.
nc - number of customers.
nv - number of vehicles.

Returns:
returns integer total tour duration. Updates tour node array as appropriate.

getId

public final int getId()

getEa

public final int getEa()

getLa

public final int getLa()

getQty

public final int getQty()

getType

public final int getType()

getWait

public final int getWait()

getLoad

public final int getLoad()

getM

public final double getM()

setId

public final void setId(int id)

setWait

public final void setWait(int wait)

setType

public final void setType(int type)

setQty

public final void setQty(int qty)

setLoad

public final void setLoad(int load)

print

public final void print()
public static final void printTour(NodeType tour[])
moveValTT

public static int moveValTT(int i,
    int d,
    NodeType tour[],
    NodeType nbrtour[],
    int time[][])

Method moveValTT computes the incremental change in the value of the travel time from the incumbent tour to the proposed neighbor tour, and computes the neighbor schedule parameters preparing for computation of penalty terms.

Parameters:
i - node position.
d - move depth.
tour - incumbent tour node array to be processed.
nbrtour - neighbor tour node array to be processed.
time - time matrix used to determine schedule.

Returns:
returns integer move value which is the resultant change in the objective function resulting from the proposed move.

See Also:
    compPens

Class PrintCalls

java.lang.Object
    +---- PrintCalls

public class PrintCalls
extends Object PrintCalls is to display on the screen initial values and rts steps as required.
Version:
    v1.1 Mar 99
Author:
    Kevin P. O'Rourke, David M. Ryer

Constructor Index
PrintCalls()

Method Index
printInitVals(int, int, int, double, String)
rtsStepPrint(int, int, int, int, int, int)

Constructors
PrintCalls

    public PrintCalls()

Methods
printInitVals

    public static void printInitVals(int nv,
                                        int iters,
                                        int
int numcycles,
double factor,
String file)

rtsStepPrint

public static void rtsStepPrint(int id,
    int i,
    int d,
    int k,
    int moveVal,
    int totNbrPen,
    int tabu,
    int numnodes)

Class PrintFlag

java.lang.Object
    |
    +-----PrintFlag

public class PrintFlag
extends Object PrintFlag contains all print out flags as boolean attributes.
Version: v1.1 Mar 99
Author: Kevin P. O'Rourke, David M. Ryer

Variable Index

cyclePrint
    print flag.
iterPrint
    print flag.
loadPrint
    print flag.
movPrint
    print flag.
startPrint
    print flag.
stepLoopPrint
    print flag.
stepPrint
    print flag.
timePrint
    print flag.
twrdPrint
    print flag.

Constructor Index

PrintFlag()
    Default PrintFlag constructor sets all to "true".
PrintFlag(boolean)
    Additional PrintFlag constructor allows specification of either "true" or "false".
Variables
movePrint

    public boolean movePrint
        print flag.
startPrint

    public boolean startPrint
        print flag.
timePrint

    public boolean timePrint
        print flag.
stepPrint

    public boolean stepPrint
        print flag.
stepLoopPrint

    public boolean stepLoopPrint
        print flag.
twrdPrint

    public boolean twrdPrint
        print flag.
cyclePrint

    public boolean cyclePrint
        print flag.
iterPrint

    public boolean iterPrint
        print flag.
loadPrint

    public boolean loadPrint
        print flag.

Constructors
PrintFlag

    public PrintFlag()
        Default PrintFlag constructor sets all to "true".
PrintFlag

    public PrintFlag(boolean set)
        Additional PrintFlag constructor allows specification of either "true" or "false".

Class ReacTabuObj

java.lang.Object
    |  
    +---ReacTabuObj
public class ReactTabuObj
extends Object

ReacTabuObj class contains the method to perform the reactive tabu search.

Version:
   v1.1 Mar 99

Author:
   Kevin P. O'Rourke, David M. Ryer

Constructor Index

ReactTabuObj()

Method Index

search(double, double, double, double, int, int, int, int, int, int, int, int, int, int, VrpPenType, int[][], PrintFlag, int, int, int, int, int, int, int, int, int, int, int, NodeType[], NodeType[], NodeType[])

   ReactTabuObj steps through iterations of the reactive tabu search.

Constructors

ReactTabuObj

   public ReactTabuObj()

Methods

search

   public static SearchOut search(double TWPEN,
                                double LDPEN,
                                double INCREASE,
                                double DECREASE,
                                int HTSIZE,
                                int CYMAX,
                                int ZRANGE,
                                int DEPTH,
                                int capacity,
                                int minTL,
                                int maxTL,
                                int tabuLen,
                                int iters,
                                int nc,
                                int numnodes,
                                VrpPenType tourPen,
                                int time[][][],
                                PrintFlag printFlag,
                                int tourCost,
                                int penTrav,
                                int totPenalty,
                                int tvl,
                                int bffTourCost,
                                int btTour,
                                int bfnv,
                                int bftier,
                                int bestCost,
                                int bestTT,
                                int bestnv,
                                int bestTime,

87
int bestTimeF,
int bestIters,
int numfeas,
NodeType tour[],
NodeType bestTour[],
NodeType bestTourF[])

ReacTabuObj steps through iterations of the reactive tabu search. This method will perform tabu
search for VRP with capacity as well as TSP without capacity.

Returns:
returns packaged output object.

Class ReadFile

java.lang.Object
   
   +-----ReadFile

public class ReadFile
extends Object
ReadFile Class reads appropriate data from a text file. Methods exist to read specific data
types (file format must be known in advance).
Version:
v1.1 Mar 99
Author:
Kevin P. O'Rourke, David M. Ryer

Constructor Index

ReadFile()

Method Index

assignInputFile(String)
assignInputFile sets up the FileInputStream.
readNextDouble(StreamTokenizer)
readNextDouble method gets the next token and returns it as a double.
readNextInt(StreamTokenizer)
readNextInt method gets the next token and returns it as an integer.
readNextString(StreamTokenizer)
readNextString method gets the next token and returns it as a string.

Constructors

public ReadFile()

Methods

assignInputFile

public static final FileInputStream assignInputFile(String filename)
assignInputFile sets up the FileInputStream.

readNextString

public static final String readNextString(StreamTokenizer st)
readNextString method gets the next token and returns it as a string.
Parameters:
   st - string tokenizer.
Returns:
    returns next string from file.

readNextDouble

    public static final double readNextDouble(SteamTokenizer st)
    readNextString method gets the next token and returns it as a double.
Parameters:
    st - string tokenizer.
Returns:
    returns next double from file.

readNextInt

    public static final int readNextInt(SteamTokenizer st)
    readNextString method gets the next token and returns it as an integer.
Parameters:
    st - string tokenizer.
Returns:
    returns next integer from file.

Class SearchOut

java.lang.Object
    +-----SearchOut

public class SearchOut extends Object SearchOut is used as a package to output multiple information from the Search method in ReacTabuObj.
Version:
    v1.1 Mar 99
Author:
    Kevin P. O'Rourke, David M. Ryer
See Also:
    Search

Variable Index
    bestCost
    bestiter
    bestny
    bestTime
    bestTour
    bestTT
    bfcost
    bfilter
    bfnv
    bftime
    bftour
    bftt
    numfeas
    penTrav
    totPenalty
    tour
    tourCost
tourPen
tvl

Constructor Index

SearchOut()
    Default constructor.
SearchOut(int, int, int, int, int, int, int, int, int, int, VrpPenType, NodeType[],
    NodeType[], NodeType[])
    Specified constructor.

Variables

totPenalty
    public int totPenalty
penTrav
    public int penTrav
tourCost
    public int tourCost
bfiter
    public int bfiter
bfcost
    public int bfcost
bfTT
    public int bfTT
bestnv
    public int bestnv
bestiter
    public int bestiter
bestCost
    public int bestCost
bestTT
    public int bestTT
bfnv
    public int bfnv
bffTime
    public int bffTime
bestTime
    public int bestTime
tvl
    public int tvl
numfeas
public int numfeas
tourPen

public VrpPenType tourPen
tour

public NodeType tour[]
bfTour

public NodeType bfTour[]
bestTour

public NodeType bestTour[]

Constructors
SearchOut

public SearchOut()
    Default constructor. Assigns all values to zero.
SearchOut

public SearchOut(int totPenalty,
    int penTrav,
    int tourCost,
    int bfiter,
    int bfCost,
    int bfTT,
    int bestnv,
    int bestiter,
    int bestCost,
    int bestTT,
    int bfnv,
    int bftime,
    int bestTime,
    int tvl,
    int numfeas,
    VrpPenType tourPen,
    NodeType tour[],
    NodeType bfTour[],
    NodeType bestTour[])

    Specified constructor. Values set as passed.

Class StartPenBestOut

java.lang.Object
    |
    +-----StartPenBestOut

public class StartPenBestOut
extends Object StartPenBestOut is used as a package to output multiple penalty information from method startPenBest.
Version:
    v1.1 Mar 99
Author:
Kevin P. O'Rourke, David M. Ryer

Variable Index

bestCost Penalty related value.
bestIter Penalty related value.
bestNv Penalty related value.
bestTime Penalty related value.
bestTour Saved tour.
bestTT Penalty related value.
bfCost Penalty related value.
bfIter Penalty related value.
bfNv Penalty related value.
bfTime Penalty related value.
bfTour Penalty related value.
bfTT Saved tour.
penTrav Penalty related value.
totPenalty Penalty related value.
tourCost Penalty related value.
tourPen Tour penalty values.

Constructor Index

StartPenBestOut()
Default constructor.
StartPenBestOut(int, int, int, int, int, int, int, int, int, int, int, VrpPenType, NodeType[],
NodeType[])
Specified constructor.

Variables
totPenalty

public int totPenalty Penalty related value.
penTrav

public int penTrav Penalty related value.
tourCost

public int tourCost

92
Penalty related value.

**bfiter**

```java
public int bfiter
    Penalty related value.
```

**bfCost**

```java
public int bfCost
    Penalty related value.
```

**bfTT**

```java
public int bfTT
    Penalty related value.
```

**bestnv**

```java
public int bestnv
    Penalty related value.
```

**bestiter**

```java
public int bestiter
    Penalty related value.
```

**bestCost**

```java
public int bestCost
    Penalty related value.
```

**bestTT**

```java
public int bestTT
    Penalty related value.
```

**bfnv**

```java
public int bfnv
    Penalty related value.
```

**bfTime**

```java
public int bfTime
    Penalty related value.
```

**bestTime**

```java
public int bestTime
    Penalty related value.
```

**tourPen**

```java
public VrpPenType tourPen
    Tour penalty values.
```

**bfTour**

```java
public NodeType bfTour[]
    Saved tour.
```

**bestTour**

```java
public NodeType bestTour[]
    Saved tour.
```
Constructors

StartPenBestOut

public StartPenBestOut()
    Default constructor. Assigns all values to zero.

StartPenBestOut

public StartPenBestOut(int totPenalty,
                        int penTrav,
                        int tourCost,
                        int bfiter,
                        int bfCost,
                        int bfTT,
                        int bestnv,
                        int bestiter,
                        int bestCost,
                        int bestTT,
                        int bfnv,
                        int bftime,
                        int bestTime,
                        VrpPenType tourPen,
                        NodeType bfTour[],
                        NodeType bestTour[])

Specified constructor. Values set as passed.

Class StartTourObj

java.lang.Object
    |
    +---->StartTourObj

public class StartTourObj
extends Object
StartTourObj class begins timing, computes an initial schedule and initial tour cost (Tour Cost = Travel time + Waiting Time + Penalty Term), computes the initial hashing values: Z(t) and thv(t), and produces a tour based on a sort of increasing avg time windows at each node. The customers are ordered by increasing avg time window value, and the nv vehicle nodes are appended to the end of the tour.

Constructor Index

StartTourObj()

Method Index

startPenBest(int, int, int, NodeType[], double, double, int, int, int, int, int, VrpPenType, int, int, int, int, int, NodeType[], NodeType[])
    startPenBest initializes "best" values and their times.

Constructors

StartTourObj

public StartTourObj()

Methods

startPenBest

public static StartPenBestOut startPenBest(int numnodes,
int tvl,
int tourLen,
NodeType tour[],
double TWPen,
double LDPPen,
int capacity,
int totPenalty,
int penTrav,
int tourCost,
VrpPenType tourPen,
int bfiter,
int bfTourCost,
int bfTT,
int bfnv,
int bestiter,
int bestCost,
int bestTT,
int bestnv,
int bestTimeF,
int bestTime,
NodeType bestTour[],
NodeType bestTourF[])

startPenBest initializes "best" values and their times. Computes cost of initial tour as tour length
with added penalty for infeasibilities.

Returns:
returns StartPenBestOut wrapper object for multiple values.

**Class TabuMod**

java.lang.Object
    |
    +-----TabuMod

public class TabuMod
extends Object TabuMod Class contains methods used in the TabuSearch. countVeh calculates the number
of vehicles used in the current tour. noCycle updates the search parameters if tour is not found in the
hashtable. cycle updates the search parameters if tour is found in the hashtable. moveValTT computes the
incremental change in the value of the travel time.

Version:
v1.1 Mar 99

Author:
Kevin P. O'Rourke, David M. Ryer

Constructor Index

TabuMod()

Method Index

countVehicles(NodeType[])
countVeh method calculates the number of vehicles used in the current tour by counting the
number of vehicle (type 2) to demand (type 1) transitions.
cycle(ValueObj, double, int, int, int, double, int, int, PrintFlag)
cycle method updates the search parameters if the incumbent tour is found in the hashing structure.
moveValTT(int, int, NodeType[], NodeType[], int[][])
Method moveValTT computes the incremental change in the value of the travel time from the.
incumbent tour to the proposed neighbor tour, and computes the neighbor schedule parameters preparing for computation of penalty terms.

**noCycle**

```java
noCycle(double, int, double, int, int, PrintFlag)
```

noCycle method updates the search parameters if the incumbent tour is not found in the hashing structure.

**Constructors**

**TabuMod**

```java
public TabuMod()
```

**Methods**

**countVehicles**

```java
public static final int countVehicles(NodeType tour[])
```

countVeh method calculates the number of vehicles used in the current tour by counting the number of vehicle (type 2) to demand (type 1) transitions.

**Parameters:**

- tour - node array to be processed.

**Returns:**

- returns integer number of vehicles used in the tour.

**noCycle**

```java
public static NoCycleOut noCycle(double DECREASE,
    int minTL,
    double mavg,
    int ssstlc,
    int tabuLen,
    PrintFlag printFlag)
```

noCycle method updates the search parameters if the incumbent tour is not found in the hashing structure.

**Parameters:**

- DECREASE - adjusting scaling factor to reduce tabu length.
- minTL - minimum tabu length.
- mavg - moving average between cycles.
- ssstlc - steps since last tabu length change.
- tabuLen - current tabu length.
- printFlag - option to print cycle information.

**Returns:**

- returns noCycleOut wrapped object.

**cycle**

```java
public static CycleOut cycle(ValueObj matchPtr,
    double INCREASE,
    int maxTL,
    int CYMAX,
    int k,
    double mavg,
    int ssstlc,
    int tabuLen,
    PrintFlag printFlag)
```

cycle method updates the search parameters if the incumbent tour is found in the hashing structure.

**Parameters:**

- matchPtr - matched information for previously found identical tour
- INCREASE - adjusting scaling factor to increase tabu length

96
maxTL - maximum tabu length
CYMAX - maximum allowable cycle frequency
k - current iteration
mavg - moving average between cycles.
ssltc - steps since last tabu length change.
tabuLen - current tabu length.
printFlag - option to print cycle information.

Returns:
returns cycleOut wrapped object.

moveValTT

public static int moveValTT(int i,
    int d,
    NodeType tour[],
    NodeType nbrtour[],
    int time[][])

Method moveValTT computes the incremental change in the value of the travel time from the
incumbent tour to the proposed neighbor tour, and computes the neighbor schedule parameters
preparing for computation of penalty terms.

Parameters:
  i - node position.
  d - move depth.
  tour - incumbent tour node array to be processed.
  nbrtour - neighbor tour node array to be processed.
  time - time matrix used to determine schedule.

Returns:
returns integer move value which is the resultant change in the objective function resulting from
the proposed move.

See Also:
  compPens

Class TimeMatrixObj

java.lang.Object
  |
  +----+TimeMatrixObj

public class TimeMatrixObj
extends Object
TimeMatrixObj contains methods to calculate the distance/time matrix based on the
problem parameters.
Version:
  v1.1 Mar 99
Author:
  Kevin P. O'Rourke, David M. Ryer

Constructor Index
TimeMatrixObj()

Method Index
readNC(String)
readNC is used to read from the first token from the input file (the number of customers (nc)).
readNV(String)
readNV is used to read from the second token from the input file (the number of vehicles (nv)).
readTSPTW(double, int, int, String, CoordType[], int[])
readTSPTW reads in the geographical coordinates and time window file and calculates the time between each node.

**readTSPTWdepot**(double, int, int, String, CoordType[], int[])
readTSPTWdepot reads in the geographical coordinates, load quantity, service time, and time window information associated with depot and customer locations from the input file.

**readTSPTWdepotUAV**(double, int, int, String, CoordType[], int[])
readTSPTWdepotUAV reads in the geographical coordinates, load quantity, min/max service times, and time window information associated with depot and target locations from the input file.

**readTSPTWrerouteUAV**(double, int, int, String, CoordType[], int[])
readTSPTWrerouteUAV reads in the geographical coordinates, load quantity, service time, and time window information associated with depot and customer locations from the input file.

**timeMatrix**(int, int, double, int, CoordType[], int[])

timeMatrix computes simple two-dimensional time/distance matrix.

**timeMatrixDepot**(int, int, double, int, CoordType[], int[])

timeMatrixDepot computes the two-dimensional array used as the "time" matrix.

**timeMatrixDepotUAV**(int, int, double, int, CoordType[], int[])

timeMatrixDepotUAV computes the two-dimensional array used as the "time" matrix.

**timeMatrixUAV**(int, int, double, int, CoordType[], int[], WindData, int[][])

timeMatrixDepot computes the two-dimensional array used as the "time" matrix.

**timeMatrixUAVreroute**(int, int, double, int, CoordType[], int[], WindData, int[][])

timeMatrixDepotUAVreroute computes the two-dimensional array used as the "time" matrix.

Constructors

**TimeMatrixObj**

public TimeMatrixObj()

Methods

**readNC**

public static int readNC(String filein)
readNC is used to read from the first token from the input file (the number of customers (nc)).

Parameters:
- filein - name of input file

Returns:
- returns nc number of customers

**readNV**

public static int readNV(String filein)
readNV is used to read from the second token from the input file (the number of vehicles (nv)).

Parameters:
- filein - name of input file

Returns:
- returns nv number of vehicles

**readTSPTW**

public static NodeType[] readTSPTW(double factor,
int nv,
int nc,
String filein,
CoordType coord[],
int s[])

readTSPTW reads in the geographical coordinates and time window file and calculates the time between each node.

Parameters:
factor - integer scaling factor used to increase precision.
nv - number of aircraft available (vehicles).
nc - number of targets/route points (customers).
filein - name of input file.
coord - blank array where coordinates will be stored upon method completion.
s - blank array where service times will be stored upon method completion.

Returns:
returns the tour array reflecting file data.

readTSPTWdepot

public static NodeType[] readTSPTWdepot(double factor,
int nv,
int nc,
String filein,
CoordType coord[],
int s[])

readTSPTWdepot reads in the geographical coordinates, load quantity, service time, and time
window information associated with depot and customer locations from the input file. This
information is returned as a tour array.

Parameters:
factor - integer scaling factor used to increase precision.
nv - number of aircraft available (vehicles).
nc - number of targets/route points (customers).
filein - name of input file.
coord - blank array where coordinates will be stored upon method completion.
s - blank array where service times will be stored upon method completion.

Returns:
returns the tour array reflecting file data.

readTSPTWdepotUAV

public static NodeType[] readTSPTWdepotUAV(double factor,
int nv,
int nc,
String filein,
CoordType coord[],
int s[])

readTSPTWdepotUAV reads in the geographical coordinates, load quantity, min/max service
times, and time window information associated with depot and target locations from the input file.
Actual service time is calculated as a random variable. This information is returned as a tour array.
This method reads in the data in degrees, minutes seconds format.

Parameters:
factor - integer scaling factor used to increase precision.
nv - number of aircraft available (vehicles).
nc - number of targets/route points (customers).
filein - name of input file.
coord - blank array where coordinates will be stored upon method completion.
s - blank array where service times will be stored upon method completion.

Returns:
returns the tour array reflecting file data.

readTSPTWrerouteUAV

public static NodeType[] readTSPTWrerouteUAV(double factor,
int nv,
int nc,
String filein,
readTSPTWrerouteUAV reads in the geographical coordinates, load quantity, service time, and time window information associated with depot and customer locations from the input file. This information is returned as a tour array. This method reads in the data in degrees, minutes seconds format. This is used to route a UAV from current position specified on the first line through a tour to its home depot.

Parameters:
- factor - integer scaling factor used to increase precision.
- nv - number of aircraft available (vehicles).
- nc - number of targets/route points (customers).
- filein - name of input file.
- coord - blank array where coordinates will be stored upon method completion.
- s - blank array where service times will be stored upon method completion.

Returns:
- returns the tour array reflecting file data.

timeMatrix

```java
public static int[][] timeMatrix(int nc,
        int gamma,
        double factor,
        int numnodes,
        CoordType coord[],
        int s[])
```

timeMatrix computes simple two-dimensional time/distance matrix.

Parameters:
- nc - number of targets/route points (customers).
- gamma - additional vehicle usage penalty (set to ZERO only).
- factor - integer scaling factor used to increase precision.
- coord - blank array where coordinates will be stored upon method completion.
- s - blank array where service times will be stored upon method completion.

Returns:
- returns the time matrix specific to the problem.

timeMatrixDepot

```java
public static int[][] timeMatrixDepot(int nc,
        int gamma,
        double factor,
        int numnodes,
        CoordType coord[],
        int s[])
```

timeMatrixDepot computes the two-dimensional array used as the "time" matrix. This time matrix contains the travel times between respective nodes, general setup for multiple depot problem.

Parameters:
- nc - number of targets/route points (customers).
- gamma - additional vehicle usage penalty (set to ZERO only).
- factor - integer scaling factor used to increase precision.
- coord - blank array where coordinates will be stored upon method completion.
- s - blank array where service times will be stored upon method completion.

Returns:
- returns the time matrix specific to the problem.

timeMatrixDepotUAV

```java
public static int[][] timeMatrixDepotUAV(int nc,
        int gamma,
        double factor,
        int numnodes,
        CoordType coord[],
        int s[])
```

timeMatrixDepotUAV
double factor,
int numnodes,
CoordType coord[],
int s[])

timeMatrixDepotUAV computes the two-dimensional array used as the "time" matrix. This time
matrix contains the hard-coded wind adjusted travel times between respective nodes.

Parameters:
nc - - number of targets/route points (customers).
gamma - - additional vehicle usage penalty (set to ZERO only).
factor - - integer scaling factor used to increase precision.
coord - - blank array where coordinates will be stored upon method completion.
s - - blank array where service times will be stored upon method completion.

Returns:
returns the time matrix specific to the problem.

timeMatrixUAV

public static int[][] timeMatrixUAV(int nc,
int gamma,
double factor,
int numnodes,
CoordType coord[],
int s[],
WindData wind,
int altitude[][])

timeMatrixDepot computes the two-dimensional array used as the "time" matrix. This time matrix
contains the wind adjusted travel times between respective nodes. This is the "primary" UAV time
matrix method for standard routing.

Parameters:
nc - - number of targets/route points (customers).
gamma - - additional vehicle usage penalty (set to ZERO only).
factor - - integer scaling factor used to increase precision.
coord - - blank array where coordinates will be stored upon method completion.
s - - blank array where service times will be stored upon method completion.
wind - - wind data object used to scale travel times.
altitude - - matrix of optimum travel altitudes between points

Returns:
returns the time matrix specific to the problem.

timeMatrixUAVreroute

public static int[][] timeMatrixUAVreroute(int nc,
int gamma,
double factor,
int numnodes,
CoordType coord[],
int s[],
WindData wind,
int altitude[][])

timeMatrixDepotUAVreroute computes the two-dimensional array used as the "time" matrix. This
time matrix contains the wind adjusted travel times between respective nodes. Node distance is
calculated from the point of origin. First vehicle returns to location specified in by second vehicle.
This is the REROUTE UAV method.

Parameters:
nc - - number of targets/route points (customers).
gamma - - additional vehicle usage penalty (set to ZERO only).
factor - - integer scaling factor used to increase precision.
coord - blank array where coordinates will be stored upon method completion.
s - blank array where service times will be stored upon method completion.
wind - wind data object used to scale travel times.
altitude - matrix of optimum travel altitudes between points

Returns:
returns the time matrix specific to the problem.

**Class Timer**

```java
java.lang.Object
    +--- Timer
```

public class Timer
extends Object
Timer Class is used to time overall computation time.
Version:
v1.1 Mar 99
Author:
    Kevin P. O'Rourke, David M. Ryer

**Variable Index**

**endTime**
end time.

**startTime**
begin time.

**totalSeconds**
duration of run.

**Constructor Index**

**Timer()**
Default constructor.

**Method Index**

**endTime()**
endTime assigns end time.

**startTime()**
startTime assigns start time.

**totalSeconds()**
total Seconds returns duration.

**Variables**

**startTime**

public long startTime
begin time.

**endTime**

public long endTime
duration of run.

**totalSeconds**

public long totalSeconds
duration of run.
Constructors

Timer

public Timer()
        Default constructor. Assigns all values to zero.

Methods

startTime

public long startTime()
        startTime assigns start time.
Returns:
        returns start time.

endTime

public long endTime()
        endTime assigns end time.
Returns:
        returns end time.

totalSeconds

public long totalSeconds()
        totalSeconds returns duration.
Returns:
        returns duration.

Class TspTwPen

java.lang.Object
              +----MTSPTWuav
                    |        +----TspTwPen

public class TspTwPen
extends MTSPTWuav tspTwPen class: Given the TW and load penalties, this procedure personalizes the penalties to the mTSPTW; Computes tourCost of tour as tour length + scaled penalty for infeasibilities.

Constructor Index

TspTwPen()

Method Index

TspTwPen(int, NodeType[], VrpPenType, double, double, int, int, int, int)
        tspTwPen method uses the TW and load penalties to computes tourCost of tour as tour length + scaled penalty for infeasibilities.

TspTwPenNormalized(int, NodeType[], VrpPenType, double, double, int, int, int, int)
        tspTwPenNormalized method uses the TW and load penalties to computes tourCost of tour as tour length + scaled penalty for infeasibilities.

Constructors

TspTwPen

public TspTwPen()
Methods

tspTwPen

public static final TsptwPenOut tspTwPen(int tourLen,
        NodeType tour[],
        VrpPenType tourPen,
        double TWPEN,
        double LDPEN,
        int totPenalty,
        int tourCost,
        int penTrav,
        int tvl)

tspTwPen method uses the TW and load penalties to computes tourCost of tour as tour length + scaled penalty for infeasibilities. This method is used with the absolute penalty factors.

Parameters:
- tourLen - tour duration.
- tour - node array to be processed.
- tourPen - current tour penalty value.
- TWPEN - time window penalty factor.
- LDPEN - load overage penalty factor.
- totPenalty - sum total penalties.
- tourCost - total tour cost.
- penTrav - travel time penalty.
- tvl - travel duration.

Returns:
- returns wrapped multiple objects.

tspTwPenNormalized

public static final TsptwPenOut tspTwPenNormalized(int tourLen,
        NodeType tour[],
        VrpPenType tourPen,
        double TWPEN,
        double LDPEN,
        int totPenalty,
        int tourCost,
        int penTrav,
        int tvl)

tspTwPenNormalized method uses the TW and load penalties to computes tourCost of tour as tour length + scaled penalty for infeasibilities. This method is uses penalty factors of one and is called when the insertion move is made. Penalty values are then comparable from iteration to iteration.

Parameters:
- tourLen - tour duration.
- tour - node array to be processed.
- tourPen - current tour penalty value.
- TWPEN - time window penalty factor (IGNORED, set to 1).
- LDPEN - load overage penalty factor (IGNORED, set to 1).
- totPenalty - sum total penalties.
- tourCost - total tour cost.
- penTrav - travel time penalty.
- tvl - travel duration.

Returns:
- returns wrapped multiple objects.
Class TsptwPenOut

java.lang.Object
  |
  +-----TsptwPenOut

public class TsptwPenOut
extends Object
TsptwPenOut is used as a package to output multiple penalty information from class
TsptwPen.
Version:
v1.1 Mar 99
Author:	Kevin P. O'Rourke, David M. Ryer

Variable Index

penTrav
Penalty related value.
totPenalty
Penalty related value.
tourCost
Penalty related value.
tvl
Penalty related value.

Constructor Index

TsptwPenOut()
Default constructor.
TsptwPenOut(int, int, int, int)
Specified constructor.

Variables
totPenalty

public int totPenalty
Penalty related value.
tourCost

public int tourCost
Penalty related value.
penTrav

public int penTrav
Penalty related value.
tvl

public int tvl
Penalty related value.

Constructors

TsptwPenOut

public TsptwPenOut()
Default constructor. Assigns all values to zero.

TsptwPenOut
public TspTwPenOut(int totPenalty,  
        int tourCost,  
        int penTrav,  
        int tvl)
    
Specified constructor. Values set as passed.

Class TwBestTTOut

java.lang.Object
   
|+++++TwBestTTOut

public class TwBestTTOut

extends Object TBestTTOut is used as a package to output multiple information from the TBestTTOut method.

Version:
    v1.1 Mar 99

Author:
    Kevin P. O'Rourke, David M. Ryer

Variable Index

bestCost  
bestTime  
bestTour  
bestTT  
bfCost  
bfTime  
bfTour  

Constructor Index

TwBestTTOut()
    Default constructor.

TwBestTTOut(int, int, int, int, int, int, int, int, NodeType[], NodeType[])
    Specified constructor.
Variables

bfCost

public int bfCost
   best tour related value.

bfTT

public int bfTT
   best tour related value.

bfnv

public int bfnv
   best tour related value.

bfiter

public int bfiter
   best tour related value.

bestCost

public int bestCost
   best tour related value.

bestTT

public int bestTT
   best tour related value.

bestnv

public int bestnv
   best tour related value.

bestiter

public int bestiter
   best tour related value.

bfTime

public int bfTime
   best tour related value.

bestTime

public int bestTime
   best tour related value.

bfTour

public NodeType bfTour[]
   best tour related value.

bestTour

public NodeType bestTour[]
   best tour related value.

 Constructors

TwBestTTOut

public TwBestTTOut()
Default constructor. Assigns all values to zero.

TwBestTTOut

public TwBestTTOut(int bfCost,
                     int bfTT,
                     int bfnv,
                     int bfiter,
                     int bestCost,
                     int bestTT,
                     int bestnv,
                     int bestiter,
                     int bftime,
                     int bestTime,
                     NodeType bfTour[],
                     NodeType bestTour[])  

Specified constructor. Values set as passed.

Class ValueObj

java.lang.Object
    |
    +----ValueObj

public final class ValueObj
extends Object ValueObj Class is used to store tour attributes in the hashtable for comparison.
Version:  v1.1 Mar 99
Author:    Kevin P. O'Rourke, David M. Ryer

Constructor Index

ValueObj(int, int, int, int, int, int)
    Specified constructor.

Method Index

equals(ValueObj)
    Overloaded equals(), check only attribute fields.
hashCode()
    Overloaded hashCode method.
toString()  
    toString changes a ValueObj to a string for use in the hashTable.

Constructors

ValueObj

public ValueObj(int fhv,
                 int shv,
                 int tourCost,
                 int tvl,
                 int twPen,
                 int loadPen,
                 int lastIter)

Specified constructor. Values set as passed.
Methods

equals

public final boolean equals(ValueObj a)
    Overloaded equals(), check only attribute fields. Do not check first two data elements to keep inline with hashCode overload.
Parameters:
    a - element compared calling object.
Returns:
    returns true if objects are equal, false otherwise.

toString

public final String toString()
    toString changes a ValueObj to a string for use in the hashTable.
Returns:
    returns concatenated String.
Overrides:
    toString in class Object

hashCode

public final int hashCode()
    Overloaded hashCode method. Note: if two objects are equal according to the equals method, then calling the hashCode method on each of the two objects must produce the same integer result. Do not checking first two data elements because of size limitations of Integer.
Returns:
    returns integer hashcode value.
Overrides:
    hashCode in class Object

Class VrpPenType

java.lang.Object
    +---- VrpPenType

public class VrpPenType
extends Object VrpPentype class provides the object structure for load and time window penalties.
Version:
    v1.1 Feb 99
Author:
    Kevin P. O'Rourke, David M. Ryer

Constructor Index

VrpPenType()
    Default constructor.
VrpPenType(int, int)
    Specified constructor.
VrpPenType(int, int, int)
    Specified constructor.

Method Index

compPens(NodeType[], int)
    compPens computes the vehicle capacity overload and time window penalties.
Constructors

VrpPenType

public VrpPenType()
    Default constructor. Assigns all values to zero.

VrpPenType

public VrpPenType(int tw,
    int ld)
    Specified constructor. Values set as passed.

VrpPenType

public VrpPenType(int tw,
    int ld,
    int nvu)
    Specified constructor. Values set as passed.

Methods

compPens

public final VrpPenType compPens(NodeType tour[],
    int capacity)
    compPens computes the vehicle capacity overload and time window penalties.
Parameters:
    tour[] - current tour used to calculate penalties.
    capacity - maximum vehicle load.
Returns:
    returns the VrpPenType object which the method was called on with updated values.

Class WindAdjust

java.lang.Object
|    +-----WindAdjust

public class WindAdjust
extends Object WindAdjust will provides the adjusted ground speed given the desired heading from
to location A to location B, and the wind heading.
Version:
    v1.1 Feb 99
Author:
    Kevin P. O'Rourke, David M. Ryer

Constructor Index

WindAdjust()

Method Index

groundSpeed(double, double, double, double)
    groundSpeed method returns the ground speed given the heading between points, the wind
heading, the wind speed, and the aircraft's airspeed.

groundSpeedAF(double, double, double, double)
    groundSpeedAF is an experimental method that uses a different formula.
Constructors

WindAdjust

    public WindAdjust()

Methods

groundSpeed

    public static final double groundSpeed(double headingAtoB,
                                     double windDir,
                                     double airSpeed,
                                     double windSpeed)

    groundSpeed method returns the ground speed given the heading between points, the wind
heading, the wind speed, and the aircraft's airspeed.

Parameters:
    headingAtoB - heading between points in degrees.
    windDir - wind heading in degrees.
    airSpeed - aircraft air speed in knots.
    windSpeed - wind speed in knots.

Returns:
    returns ground speed in knots.

groundSpeedAF

    public static final double groundSpeedAF(double headingAtoB,
                                     double windDir,
                                     double airSpeed,
                                     double windSpeed)

    groundSpeedAF is an experimental method that uses a different formula. It has not been validated
and is not used. Designed to return the ground speed given the heading between points, the wind
heading, the wind speed, and the aircraft's airspeed.

Parameters:
    headingAtoB - heading between points in degrees.
    windDir - wind heading in degrees.
    airSpeed - aircraft air speed in knots.
    windSpeed - wind speed in knots.

Returns:
    returns ground speed in knots.

Class WindData

java.lang.Object
    |
    +---- WindData

public class WindData
extends Object
WindData Class provides attributes to store wind direction, velocity, and UAV velocity for
a given altitude plus methods to read in wind information from a file and manually manipulate wind data.
Version:
    v1.1 Mar 99
Author:
    Kevin P. O'Rourke

111
**Variable Index**

*acSpeed*  
double aircraft speed at the associated altitude level.

*altitude*  
integer value of the associated altitude level.

*bands*  
double Number of altitude level bands.

*bearing*  
double wind bearing at the associated altitude level.

*speed*  
double wind speed at the associated altitude level.

**Method Index**

*getACspeed(int)*  
getACspeed returns aircraft (UAV) speed for the specified band.

*getAltitude(int)*  
getAltitude returns actual altitude for the specified band.

*getBands()*  
getBands returns number of altitude bands (wind tiers).

*getBearing(int)*  
getBearing returns wind bearing for the specified band.

*getSpeed(int)*  
getSpeed returns wind speed for the specified band.

*readWind(String)*  
readWind method reads wind data from a file and returns a WindData object.

*setACspeed(int, int)*  
setACspeed sets only the UAV speed information only for a particular band.

*setACspeedAll(int)*  
setACspeedAll sets only the UAV speed information.

*setAll(int, int, double, double, int)*  
setAll method sets all related information for a particular band.

*setAltitude(int, int)*  
setAltitude sets only the altitude information for a particular band.

*setBearing(int, double)*  
setBearing sets only the bearing information only for a particular band.

*setSpeed(int, double)*  
setSpeed sets only the wind speed information only for a particular band.

*setWind(int, int, double, double)*  
setWind method sets only wind related information for a particular band.

**Variables**

*altitude*  
protected int altitude[]  
integer value of the associated altitude level.

*bearing*  
protected double bearing[]  
double wind bearing at the associated altitude level.

*speed*  
protected double speed[]  
double wind speed at the associated altitude level.

*acSpeed*
protected int acSpeed[]
double aircraft speed at the associated altitude level.

bands

protected int bands
double Number of altitude level bands.

Methods

setWind

public void setWind(int level, int altitude, double bearing, double speed)
setWind method sets only wind related information for a particular band.
Parameters:
    altitude - actual altitude level for this band.
    bearing - wind bearing for this band.
    speed - wind speed for this band.

setAlt

public void setAlt(int level, int altitude)
setAltitude sets only the altitude information for a particular band.
Parameters:
    level - altitude band array index number.
    altitude - actual altitude level for this band.

setBearing

public void setBearing(int level, double bearing)
setBearing sets only the bearing information only for a particular band.
Parameters:
    level - altitude band array index number.
    bearing - wind bearing for this band.

setSpeed

public void setSpeed(int level, double speed)
setSpeed sets only the wind speed information only for a particular band.
Parameters:
    level - altitude band array index number.
    speed - wind speed for this band.

setACspeed

public void setACspeed(int level, int acSpeed)
setACspeed sets only the UAV speed information only for a particular band.
Parameters:
    level - altitude band array index number.
    acSpeed - aircraft (UAV) speed for this band.

setAll

public void setAll(int level,
int altitude,
double bearing,
double speed,
int acSpeed)

setAll method sets all related information for a particular band.

Parameters:
altitude - actual altitude level for this band.
bearing - wind bearing for this band.
speed - wind speed for this band.
acSpeed - aircraft (UAV) speed for this band.

setACSpeedAll

public void setACSpeedAll(int acSpeed)
setACSpeedAll sets only the UAV speed information. It assigns the same speed for all bands.

Parameters:
acSpeed - aircraft (UAV) speed for every band.

getAltitude

public int getAltitude(int level)
getAltitude returns actual altitude for the specified band.

Parameters:
level - altitude band array index.

Returns:
returns actual altitude.

getBearing

public double getBearing(int level)
getBearing returns wind bearing for the specified band.

Parameters:
level - altitude band array index.

Returns:
returns wind bearing.

getSpeed

public double getSpeed(int level)
getSpeed returns wind speed for the specified band.

Parameters:
level - altitude band array index.

Returns:
returns wind speed.

getACspeed

public int getACspeed(int level)
getACspeed returns aircraft (UAV) speed for the specified band.

Parameters:
level - altitude band array index.

Returns:
returns aircraft (UAV) speed.

getBands

public int getBands()
getBands returns number of altitude bands (wind tiers).

Returns:
returns number of altitude bands (wind tiers).

readWind
public static WindData readWind(String filein)

readWind method reads wind data from a file and returns a WindData object.

Parameters:
  filein - name of wind data file.

Returns:
  returns a WindData object
Appendix D. Additional References


Vita

Captain Kevin P. O’Rourke was born on 22 July 1968 in Los Angeles County, California. He graduated from Bel Air High School, Bel Air, Maryland, in 1986 and entered undergraduate studies in engineering at The Pennsylvania State University in State College, Pennsylvania. While attending Penn State, he enrolled in the Air Force Reserve Officer Training Corps. He graduated with a Bachelor of Science degree in Electrical Engineering and was concurrently commissioned in May 1990.

Captain O’Rourke completed Undergraduate Space Training at Lowry AFB, Colorado in August 1991 and was assigned as an orbit analysis officer at Falcon AFB Colorado (subsequently renamed Schriever AFB). In August 1994, he attended Undergraduate Missile Training at Vandenberg AFB, California, and was next assigned as a Peacekeeper ICBM Crew Commander at F. E. Warren AFB, Wyoming.

In August 1997, Captain O’Rourke entered the School of Engineering, Air Force Institute of Technology.

Permanent Address: 4210 Cromwell Court
Colorado Springs CO 80906
# Dynamic Unmanned Aerial Vehicle (UAV) Routing with a Java-Encoded Reactive Tabu Search Metaheuristic

## Abstract

In this paper we consider the dynamic routing of unmanned aerial vehicles (UAVs) currently in operational use with the US Air Force. Dynamic vehicle routing problems (VRP) have always been challenging, and the airborne version of the VRP adds dimensions and difficulties not present in typical ground-based applications. Previous UAV routing work has focused on primarily on static, pre-planned situations; however, scheduling military operations, which are often ad-hoc, drives the need for a dynamic route solver that can respond to rapidly evolving problem constraints. With these considerations in mind, we examine the use of a Java-encoded metaheuristic to solve these dynamic routing problems, explore its operation with several general problem classes, and look at the advantages it provides in sample UAV routing problems. The end routine provides routing information for a UAV virtual battlespace simulation and allows dynamic routing of operational missions.

## Subject Terms


---

### Table of Contents

<table>
<thead>
<tr>
<th>Role</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AGENCY USE ONLY</strong></td>
<td>Leave blank</td>
</tr>
<tr>
<td><strong>REPORT DATE</strong></td>
<td>March 1999</td>
</tr>
<tr>
<td><strong>REPORT TYPE AND DATES COVERED</strong></td>
<td>Master's Thesis</td>
</tr>
<tr>
<td><strong>FUNDING NUMBERS</strong></td>
<td></td>
</tr>
<tr>
<td><strong>PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</strong></td>
<td>Air Force Institute of Technology</td>
</tr>
<tr>
<td></td>
<td>2950 P Street</td>
</tr>
<tr>
<td></td>
<td>Wright Patterson AFB OH 45433-7765</td>
</tr>
<tr>
<td><strong>SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)</strong></td>
<td>Lt Col Mark A. O'Hair, Chief, Systems Integration Division (DOM)</td>
</tr>
<tr>
<td></td>
<td>Unmanned Aerial Vehicle Battlelab</td>
</tr>
<tr>
<td></td>
<td>1003 Nomad Way Ste 107</td>
</tr>
<tr>
<td></td>
<td>Eglin AFB FL 32542-6867</td>
</tr>
<tr>
<td><strong>DISTRIBUTION AVAILABILITY STATEMENT</strong></td>
<td>Approved for public release; distribution unlimited.</td>
</tr>
<tr>
<td><strong>NUMBER OF PAGES</strong></td>
<td>132</td>
</tr>
<tr>
<td><strong>PRICE CODE</strong></td>
<td>UL</td>
</tr>
</tbody>
</table>