Abstract

Riego, Dustin Bowers: Optimization of Vehicle Routes for a Third Party Logistics Provider. (Under the direction of Professor Russell E King)

This work addresses the problem of minimizing the number of vehicles required to service a weekly delivery schedule for a transportation company. The characteristics of this specific problem require each delivery to originate from the network's primary depot. Therefore, the overall mileage travelled by the entire fleet is the same for all feasible solutions. The routes generated are constrained by a maximum allowable weekly mileage and the fleet is homogenous and located at the primary depot.

Improvements to an algorithm developed in previous research are introduced in this work. These improvements are highlighted by the implementation of four separate selection criteria which are used to determine the best job to add to a route during route construction. Additionally, costs are optimized by evaluating backhaul carrying capabilities of the solution routes generated. 1000 test problems are generated for each of six job size scenarios. The new algorithm developed by this work shows improvement over previous algorithms in both computation time and overall performance.

Optimization of Vehicle Routes for a Third Party Logistics Provider

By

DUSTIN BOWERS RIEGO

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

INDUSTRIAL ENGINEERING

Raleigh

2005

APPROVED BY:

_____ _____

_____

Chair of Advisory Committee

Biography

Dustin B. Riego was born on October 31st, 1980 in Raleigh, NC. He graduated high school from Wake Forest – Rolesville High School in 1999 and pursued an undergraduate degree in Industrial Engineering at North Carolina State University.

After graduating cum laude in May 2003, Dustin continued his education in August 2003 pursuing a Master of Science degree, also in Industrial Engineering at North Carolina State University. For five years he worked under the direction of Dr. Russell E. King and Dr. Thom J. Hodgson as a Research Assistant. In his spare time he enjoys reading, web development and various outdoor activities.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

1.1 Background

A logistics company, referred to hereafter as TMC, provides transcontinental transportation services for a client. The client, ACMI, utilizes a distribution network consisting of several Forward Distribution Centers (FDC's) that receive shipments from a single National Distribution Center (NDC).

Each FDC requires weekly shipments of product from the NDC. The number of shipments to each FDC during the week depends on the product demand at the FDC, and typically does not exceed five per week. Shipments to an FDC are full truckload; therefore a truck can only service one FDC per trip. The roundtrip from the NDC to the FDC represents a job on the schedule.

Several characteristics define each job. A job's processing time includes the round trip travel time between the NDC and the FDC based on distance and average speed, and loading and unloading times at each distribution center. A job has a specified time window in which to deliver the load to the FDC defined by a given earliest delivery time and latest delivery time. For some jobs this time window may be degenerate, i.e. the earliest delivery time equals the latest delivery time. In either case, the time window defines an early start time and a late start time, i.e. time a truck must depart the NDC in order to deliver within the window.

In addition to the regular weekly job schedule, TMC would like to take advantage of backhaul opportunities. Backhauls are delivery jobs along or in close proximity to a truck's return route to the NDC. For a backhaul, the truck will travel from the FDC to a pick-up location and transfer the load to a delivery location, and then return to the NDC. Backhauls help decrease the number of "deadhead" miles on a route, or miles traveled with an empty trailer.

1.2 Problem Statement

The objective of this research is to develop a scheduling system capable of generating weekly truck route schedules that minimize TMC's costs while maximizing service to their client. Regarding the truck routes, TMC's principle concerns include:

- Minimizing the number of trucks needed to complete the job schedule

- Providing attractive routes to the drivers, which typically range from 4,000 – 5,000 miles per week
- Minimizing the number of deadhead miles by picking up backhauls

In addition to the weekly route scheduler, this research presents alternatives for the handling of "one-time" jobs. TMC typically makes changes to their route schedules once a month. If, however, an extra delivery must be made to a particular FDC, TMC would like to know if they can make the extra delivery without altering their entire route schedule. Also, the one-time job approach would provide a method for handling the uncompleted jobs on a disrupted route. Occasionally, a truck may breakdown or for some other reason be unable to complete its weekly route. If the jobs left on that route can be rescheduled on other routes, then TMC can maintain a high service level to their client without being forced to increase costs significantly.

As a final point of focus, this research examines the balance between the size of the vehicle fleet and the number of extra jobs that can be picked up by TMC. Clearly, the fewer number of trucks used in the schedule, the fewer and smaller the time gaps are between jobs, translating into fewer opportunities for backhauls. However, carrying fewer backhauls means more deadhead miles, an inefficient use of resources, and the potential loss of revenue. If the scheduling system can generate a set of route schedules that requires only one or two more trucks than minimal, but allows significantly more opportunities for backhauls, then TMC's costs can be optimized.

Chapter 2 Literature Review

The problem under consideration in this work is related to the traditional Vehicle Routing Problem with Time Windows (VRPTW). Solomon and Desrosiers (1988) and Desrochers *et al*. (1988) provide literary surveys on previously developed heuristics and exact methods for solving VRPTW, which is a NP-hard problem. Fisher *et al*. (1997) divided these heuristics and exact methods into four subcategories: (i) dynamic programming approaches; (ii) column generation approaches; (iii) methods based on Lagrangian decomposition; and (iv) K-tree based methods. Following the work of Fisher *et al*. (1997), three solution methods were capable of optimally solving problems with up to 100 customers: approaches based on column generation by Desrochers *et al*. (1992), Lagrangian relaxation/variable splitting by Fisher *et al*. (1997), and the K-tree approach by Fisher *et al*. (1997).

While the solution methods discussed have reached desirable performance levels for the VRPTW, our problem differs significantly enough to render previous solution methods invalid for our use. Most notably, our problem presents distinctive differences in route structure and objective. With the VRPTW, a route typically consists of a single departure from the depot, several intermediary visits and then a return to the depot. Conversely, a route in our problem consists of many departure-return trips from the depot to the various delivery sites. Also, solution methods for the VRPTW focus primarily on minimizing total travel times while the objective of this research is to minimize the number of routes and, therefore, the number of tractors.

A subclass of the VRPTW, the Rollon-Rolloff Vehicle Routing Problem (RRVRP) emerged from a set of sanitation collection routing problems. In the RRVRP, large trailers positioned at construction sites or other similar dump sites are transported by tractors from the disposal facility to their service location. Because the trailers are full truckloads, the tractors can only service one dump site per trip. Bodin *et al*. (2000) presented four trip type specifications for the RRVRP. The route structure presented by this work closely mimics the structure represented by the Type 1 Trips in Bodin *et al*.'s (2000) specifications. The Type 1 Trip defines the case where a tractor departs from the disposal facility, picks up the trailer at the service location, and then returns to the disposal facility. Concentrating solely on the travel behavior and ignoring the details of

the load carrying, this is the exact situation described in our problem. Bodin *et al*. (2000) also identified minimizing the number of vehicles as a secondary objective, though they maintain minimizing total travel time as their primary objective. The aspects of our problem not addressed by the RRVRP are the added complexity of delivery time windows, the incorporation of backhaul opportunities, and minimizing the number of routes or trucks as the primary objective.

Bombien (2001) first investigated the problem discussed in this work, and proposed a randomized constructive algorithm as an approach to solving this problem. Labiad (2002) simplified Bombien's algorithm, and created four variants of the algorithm: the forward heuristic, the reverse heuristic, the random forward heuristic, and the parallel heuristic. The forward and random forward heuristics emulated Bombien's constructive approach of building routes one by one. The first job on a route is either selected randomly or according to some selection criteria. Subsequent jobs on a route are then chosen according to some predetermined criteria such as longest processing time. When maximum route constraints are met, a new route is started and the process is repeated. Once all of the jobs have been assigned to a route, the algorithm is finished. Labiad's reverse heuristic works in a similar way, except it begins at the end of the routes rather than the beginning. The parallel heuristic builds multiple routes at a time, beginning with some given number of routes.

The research by Bombien and, subsequently, Labiad did not address the problems associated with interrupted routes or inserted, one-time only jobs. Bombien included backhauls in his scheduling routine, whereas Labiad's more compact algorithm did not include the scheduling of backhauls.

Chapter 3 Basic Route Scheduling

3.1 Problem Description

The foundation for this research is the development of a system to generate a weekly truck route schedule that satisfies a set of deliveries between the NDC and FDCs for TMC's client. The schedule must meet several industry regulations as well as TMC's own performance objectives. These requirements comprise the constraints of the problem and include, per route: maximum weekly mileage, maximum weekly time on the road, and maximum tractor load.

A list of weekly deliveries provided by AMCI serves as the input for the route scheduling heuristic. The notation that follows represents the defining characteristics of each job.

M_j = mileage of job j

FDC_j = Index of the delivery's FDC site from a list of FDC sites

$ChangeIndicator_j$ = Indicates how the job can be moved (None, Time Window, Anytime)

E_j = earliest departure time of job j

D_j = normal departure time of job j

L_j = latest departure time for job j

C_j = travel time from NDC to FDC for job j

R_j = return time from FDC to NDC for job j

P_j = Processing time for job j

$ExIndicator_j$ = Indicates status of an exclusion period (time window a job cannot be delivered)

ExA_j = beginning of exclusion period for job j

ExB_j = end of exclusion period for job j

U = set of unrouted jobs

In order to clearly describe the steps in the heuristic, the following notation is used.

k = current route

i_k = current leg assignment (on the route k)

idx_{ki} = Index of job scheduled for leg i on route k

$\boldsymbol{S_{ki}}$ = start time of i^{th} leg on route k

$\boldsymbol{C_{ki}}$ = completion time of i^{th} leg on route k

$\boldsymbol{M_{ki}}$ = mileage of i^{th} leg on route k

$\boldsymbol{P_{ki}}$ = processing time of i^{th} leg on route k

3.2 Earliest Forward Heuristic

The heuristic introduced by Bombien (2001) and the modifications made by Labiad (2002) provide the foundation for the heuristic developed in this work. The process begins by first setting an upper bound on the problem equal to the number of jobs to be scheduled. Clearly, each job must take less than 168 hours to complete and require fewer than 5500 miles of traveling in order for a complete route schedule to be feasible. Therefore the worst case solution requires each job to have its own truck and route.

Next, all jobs are stored in the set, $\boldsymbol{U}$, of unrouted jobs. A job from this set is chosen at random to start the first route and is removed from set U. The chosen job is assigned to the first leg on the route. Following the assignment of the first job on the route, the heuristic begins identifying jobs to complete the route. The remaining steps continue until all unrouted jobs have been assigned to a route.

From the set of unrouted jobs, a subset of candidate jobs is selected. The candidate jobs are those unrouted jobs that could feasibly fit on route k. The feasibility of job j depends on the current total mileage covered by route k, the total processing or driving time on route k, and the completion time of the last scheduled leg on route k. If the processing time of job j or the mileage associated with job j would make the respective route totals exceed the defined maximums, the job is not included in the feasible set of jobs.

In addition to the basic mileage and processing time feasibility constraints, the departure time window for job j must be considered. The best start time is defined as the earliest time a truck can depart for delivery without modifying previously scheduled route legs. If a job's earliest departure time occurs before the completion time of the last scheduled leg on route k and the latest departure time occurs after, then the candidate job may be scheduled on the route with a departure time equal to the completion time of the preceding scheduled leg. If the candidate job's earliest departure time occurs after the

completion time of the preceding scheduled leg on the route, then the best start time for the candidate job is its earliest departure time. One of the improvements to previous heuristics introduced by this work is a subtle, yet significant modification to the calculation of the best start time for each job relative to the current route *k*. This calculation is discussed in more detail in Section 3 of this chapter.

Once the subset of feasible candidate jobs has been filled, the next job to add to the route is chosen based on some selection criterion. In the implementation, this selection criterion can be set by the user, or the user can tell the system to randomly choose a selection criteria. The selection criterion options are: longest processing time, shortest processing time, earliest start time, and random. Once a job has been chosen, the job is added to the route as a new route leg, and the process of identifying candidate jobs is repeated.

As the process continues, if there are no feasible candidate jobs to go on a route, a new route is initiated. Once again, a job is chosen randomly to start the route, and the process of identifying candidate jobs for the new route follows the steps discussed previously.

The algorithm can be summarized by the following steps.

Step 0: Set T = number of jobs to be scheduled.

Step 1: Set $i = 1$, $k = 1$

Step 2: From the set of unrouted jobs U, randomly select a job j to start route k and assign the job to the route as route leg i. Remove the chosen job j from the set of unrouted jobs.

Step 3: Identify candidate jobs to go on the current route, k.

Step 4: Select the next job j to go on the route from the subset of candidate jobs.

a) Increment i by 1, add the selected job j to route k as leg i, and remove it from the list of unrouted jobs.

b) If there are no feasible jobs, increment k by 1 and return to Step 2.

Step 5: Repeat Steps 2-4 while number of unrouted jobs > 0.

3.3 Improvements

While the scheduling routines in the previous research calculated the best departure time for a candidate job in a similar manner, there are some enhancements introduced by this research. First, in previous implementations, some jobs were eliminated as candidates because of the "wrap-around" scheduling effect of the heuristic. When the first job on a route is assigned, the start to the route's one-week (168 hour) limitation on processing time begins with the route start time, or the start time of the first job on the route, rather than hour zero. Therefore, the route end time, or the hour after which jobs can no longer be added to the route, is 168 plus the route start time. In effect, the time period between hour 168 and the route end time represents the part of the week between hour zero and the start time of the route's first job where the route has wrapped around on itself. Because the candidate job selection process did not account for this "wrap-around" effect, unrouted jobs that could feasibly fit on a particular route were being eliminated from consideration.

In order to correctly identify all possible candidate jobs, the start times for the unrouted jobs must be adjusted for the "wrap-around" effect. To do this, the start time for the unrouted job is compared with the route start time. If the unrouted job's start time is less than the route start time, then 168 is added to the unrouted job's start time. If the unrouted job's start time is greater than the route start time, then the unrouted job's start time is not modified. These "adjusted" start times are then used in the candidate selection process and will allow those jobs that start in the "wrap-around" period to be considered if they also fulfill the other criteria for candidacy.

Another enhancement to the heuristic introduced by this research expedites the search for the best solution. One of the advantages of a randomized heuristic is that if you can iterate through many solutions very quickly, you are more likely to find or approach the optimal solution. If you can identify a bad solution before you finish developing it, you can skip the process of building the rest of the solution and move on to the next solution. Hence, in our implementation of the heuristic, once the current route construction has reached the number of routes in the best solution, the current route construction is aborted and a new route construction begins. If a route construction never reaches the current best solution, then it becomes the current best solution.

Chapter 4 Backhaul Scheduling

4.1 Problem Description

During the return trip from an FDC to the NDC, trailers typically are either empty or carrying empty totes from the FDC. TMC would like to make better use of their trailers during the return trip by attempting to schedule available backhauls. Backhauls provide opportunities for additional revenue without a substantial increase in costs. In practice, TMC can retrieve information regarding available backhauls from a brokerage service and then manually search through their schedules to determine if one of their trucks is available to carry the backhaul. It is beyond the scope of this work to explore the advantages and disadvantages of automating the backhaul scheduling process. However, it is important that TMC have the capability to generate weekly truck route schedules that are flexible enough to take advantage of backhaul opportunities. Therefore, a piece of this work focuses on systematically scheduling a list of available backhauls and then evaluating the route schedule's availability for carrying backhauls.

In this work, two characteristics define a backhaul job: location and load size. All of the sites that offer backhauls are stored in a list. For each backhaul job, the location of the backhaul pickup is represented by the site's index from a list of backhaul sites. Using the site indices, mileage data can be obtained for the following situations: backhaul site to backhaul site travel, FDC site to backhaul site travel, and backhaul site to NDC site travel.

Because there is a maximum load (in pounds) that can be carried at once by a trailer, each backhaul job must have an associated load size. A trailer can carry loads from multiple backhaul locations, but the aggregate load from all pickups must be less than the maximum load allowed on a single trailer. For the purposes of this research, all backhaul loads are carried back to a location negligibly close to the NDC. Preserving the notation introduced in Chapter 3, the characteristics of a backhaul job are described by the notation below.

$\boldsymbol{B}$ = Set of unscheduled backhaul jobs

$\boldsymbol{m}$ = current backhaul job

$\boldsymbol{F_m}$ = Backhaul site location ID for backhaul job m

$\boldsymbol{W_m}$ = Backhaul load size for backhaul job m

$\boldsymbol{T_m}$ = Travel mileage to backhaul job m

$\boldsymbol{Y_m}$ = Travel mileage to NDC from backhaul job m

$\boldsymbol{Q_m}$ = Backhaul start time (time tractor leaves destination that precedes backhaul job m)

4.2 Backhaul Packing Heuristic

The backhaul scheduling heuristic is a greedy approach that iterates through the set B of unrouted backhauls, in the order that they are passed to the routine, and schedules the job on the first route leg that will accept it. If no route leg can carry job m, then job m is considered a lost opportunity and the heuristic moves on to the next backhaul job in B. The details of the heuristic follow.

To begin, the first backhaul job, m, is selected from the list of unscheduled backhaul jobs, B. Next, we search for a route leg capable of carrying the backhaul job, beginning with the first leg (i=1) of the first route (k=1) in the route schedule. There are four primary checks that a backhaul job must pass to determine if the job will fit on a candidate route leg. First, the addition of backhaul job m must not cause the overall load carried by the trailer servicing leg i of route k to exceed the maximum permissible load. This constraint is especially important when a trailer is picking up backhaul loads from multiple sites.

The second check requires that the added mileage for traveling from the preceding site to the backhaul site, and then from the backhaul site to the NDC does not cause the mileage traveled over all legs of the route to exceed the maximum allowable weekly travel mileage. In a similar vain, the third check requires that the additional processing time required to pick up backhaul m does not cause the total processing time over all legs of the route to exceed the maximum allowable weekly travel time. The additional processing time for a backhaul job includes travel time to the backhaul site, travel time from the backhaul site to the NDC, and loading time. The fourth and final check ensures that the backhaul can be picked up, loaded, and returned to the NDC before the next leg on the route begins, including the turnaround time.

It is important to note that the travel mileage and travel time characteristics of the backhaul job change depending on the route leg under consideration for scheduling the job. For example, the FDC receiving a delivery on the first leg of a route may be

different than the FDC receiving a delivery on the second leg of the route. Therefore, the distance between the site of the backhaul job and the first FDC will differ from the distance between the site of the backhaul job and the second FDC. This attribute requires a dynamic calculation of the backhaul job's travel mileage ($\boldsymbol{T_m}$ and $\boldsymbol{Y_m}$) and travel time relative to the candidate route leg. The dynamic calculation prevents us from using a simpler, quicker check of the sizes of the gaps between route legs. Additionally, there are backhaul site-to-backhaul site and FDC-to-backhaul site travel combinations that are considered practically infeasible and are denoted by $\boldsymbol{T_m}$ or $\boldsymbol{Y_m}$ values equal to zero. If the heuristic reads a zero mileage value it will consider the route *k,* leg *i* an infeasible candidate for backhaul job *m* and move on.

If backhaul job *m* passes all four of these checks for route *k*, leg *i*, job *m* is added to the route leg. The characteristics of the route leg such as leg mileage and leg processing time are updated to reflect the addition of the new backhaul job and the backhaul job is marked as scheduled. The heuristic then jumps to the next job in the unscheduled backhaul job list and repeats the process. If a backhaul job does not pass the checks for a particular leg, the next leg on the route is considered until there are no more route legs on the route to consider. The process is then repeated for the rest of the routes in the schedule until a capable route leg is found. If no leg on any route can feasibly carry the backhaul job, then the backhaul job is considered a lost opportunity and the heuristic moves to the next job in the set of unscheduled backhaul jobs, *B*.

The steps of the backhaul scheduling heuristic are summarized by the following:

Step 1: Set $m = 1$, the first unscheduled backhaul job in *B*.

Step 2: Set $k = 1$, $i = 1$.

Step 3: Test availability of route *k*, leg *i* for job *m*.

Step 4: If job *m* fits on leg *i* of route *k*, schedule job *m*. Increment *m* by 1 and return to Step 2.

Step 5: Increment *i* by 1 if $i <$ number of legs on route *k*. Otherwise increment *k* by 1. Return to Step 3.

Step 6: For each *m* repeat Steps 3-5 until all routes have been examined for availability. If no route can be found for job *m*, increment *m* by 1 and return to Step 2. Repeat for all $m \in B$

Chapter 5 Job Insertion

5.1 Problem Description

Another scheduling concern addressed by this work is job insertion. Job insertion is the process of inserting a primary job, or delivery to an FDC, into an existing route schedule. Job insertion differs from backhaul scheduling in that a new leg is added to an existing route rather than a route leg merely being extended. There are three basic scenarios that may require the job insertion technique: scheduling non-recurring FDC deliveries, schedule repair, and schedule improvement.

For the basic scheduling procedure, TMC maintains a list of weekly deliveries. This list of deliveries generally remains constant for a given period, during which TMC would like to maintain the same truck route schedule. However, there may be an occasional extra delivery that needs to be made. Rather than regenerating the entire route schedule to include the extra delivery, TMC would like to be able to add the job to an existing route if available. A discussion of the methods explored to accomplish single job insertion follows the explanation of the other two job insertion scenarios.

Both schedule repair and schedule improvement are procedures for identifying scheduled jobs that need to be rescheduled, albeit for different reasons. In schedule repair mode, jobs scheduled on a route serviced by a tractor that is temporarily out of service are marked for rescheduling. In schedule improvement mode, single routes can be marked by either the user or by the system according to some performance objective as undesirable routes. The associated jobs on the undesirable routes are then marked for rescheduling and passed to the job insertion procedure.

5.2.1 Job Insertion Approach

Conceptually, job insertion involves the basic steps of checking for a large enough gap of idle time on a route, determining the feasibility of adding the respective job to the route in the available gap, and then adding the job if feasible. However, due to the nature of the constraints, the actual implementation of job insertion requires a little more than a simple check for fit. The two methods investigated by this work are iterative, brute force techniques that can be executed quickly using existing technology. The methods differ only in how they evaluate each particular gap between route legs. Thus, the methods are

described concurrently and their differences noted where applicable. The two methods are Simple Gap Insertion and Push-Pull Gap Insertion. Typically, the Simple Gap Insertion routine executes more quickly than the Push-Pull Gap Insertion routine; however, Push-Pull Gap Insertion is generally more effective.

5.2.2 Job Insertion Heuristic

In one sense, job insertion can be interpreted as an alternative approach to scheduling routes. Hence, the input job data is exactly the same as for the Earliest Forward Heuristic discussed earlier in this work. The defining characteristics of the input schedule and the output route schedule are represented in the same way as before:

M_j = mileage of job j
FDC_j = Index of the delivery's FDC site from a list of FDC sites
$ChangeIndicator_j$ = Indicates how the job can be moved (None, Time Window, Anytime)
E_j = earliest departure time of job j
D_j = normal departure time of job j
L_j = latest departure time for job j
C_j = travel time from NDC to FDC for job j
R_j = return time from FDC to NDC for job j
$ExIndicator_j$ = Indicates status of an exclusion period (time window a job cannot be delivered)
ExA_j = beginning of exclusion period for job j
ExB_j = end of exclusion period for job j
U = set of unrouted jobs to be inserted

In addition to the input job data, job insertion also requires a route schedule as input. The routes in a particular route schedule can be represented by the following notation:

k = current route
i = current leg assignment (on the route k)
idx_{ki} = Index of job scheduled for leg i on route k

$\boldsymbol{S_{ki}}$ = start time of i^{th} leg on route k

$\boldsymbol{C_{ki}}$ = completion time of i^{th} leg on route k

$\boldsymbol{M_{ki}}$ = mileage of i^{th} leg on route k

$\boldsymbol{P_{ki}}$ = processing time of i^{th} leg on route k

$\boldsymbol{RS_k}$ = start time of route k (S_{ki} for i=1 on route k)

$\boldsymbol{RC_k}$ = completion time of route k (S_{ki} + 168)

Similar to the backhaul scheduling procedure, the job insertion heuristic is a greedy approach where the first job in the list of unscheduled job is assigned to the first feasible route gap and so on through the list of unscheduled jobs. If no feasible assignment can be made, then the job remains marked as unscheduled. The handling of jobs that cannot be inserted is beyond the scope of this work and is left up to the user.

The procedure begins by selecting the first job, j, from the set U of unscheduled jobs. Next we set k equal to the first route in the route schedule. Before we begin checking for available gaps between route legs, though, we compare the route's characteristics with some of our constraints. The additional mileage and processing time from adding job j to route k can be calculated quickly and may eliminate route k as a possibility. Because this is an iterative process, the computation time needed to fully execute the procedure is a function of the number of jobs to be inserted, the number of routes and legs on the routes, and the particular availability of the resources serving each route. For example, an unscheduled job that has a wide time window may be easier to fit on a route with several large gaps. If this flexible route appears early in the list, then the procedure will end quickly. An example that may cause the procedure to drag can be similarly conceived. By eliminating infeasible routes prior to more complex computations, some overall computation time can be saved.

If route k passes the two constraint checks and k is to be considered, then we begin the process of iterating through the legs on route k. The current route leg, i, is set to the first leg on route k. At this point, the two insertion methods diverge in the way they inspect the gaps between route legs.

Simple Gap

To determine the feasibility of a gap between route leg *a* and route leg *b*, the simple gap method evaluates the gap without attempting to move the legs that surround it. For example, if route leg *a* is scheduled to end at time t_a and route leg *b* is scheduled to begin at time t_b then job *j* must be able to begin and end between times t_a and t_b in order for the gap to be feasible.

Push-Pull Gap

The simple gap method executes quickly, however it fails to take advantage of the implicit flexibility of scheduling jobs with time windows. The push-pull method better exploits the inherent flexibility provided by the time window characteristics of jobs already assigned to the route legs. During the basic scheduling process, each leg's start time is set according to the assigned job's earliest start time relative to its time window and the completion time of the leg that precedes the current leg. Using the push-pull method, the start times of all route legs following route leg *i* are moved to their latest possible start time according to the leg's assigned job.

The process begins with the last leg on the route. The last leg's start time is set to either its associated job's latest start time or the latest time it can start and finish before it reaches the route's completion time, RC_k. Then, iterating backwards through the legs on route *k* through leg *i+1*, each leg's start time is set to either its associated job's latest start time or the latest time it can start and still finish before the start of the leg immediately following. After all of the route legs following leg *i* have been moved, the gap's feasibility as a sufficient fit for the unscheduled job *j* is evaluated.

Once a feasible gap on route *k* has been detected by either method, job *j* is inserted into the route as leg *i+1* and the route leg index on subsequent legs is incremented by one. The routine ends for the job *j*, and returns to the set of jobs *U*, to check for any remaining unscheduled jobs. All jobs in set *U* for which no feasible gap can be found are returned and handled by the user.

It is important to note here that, contrary to the way they are described, the two gap evaluation methods are not implemented concurrently in the scheduling system. For example, a gap is not evaluated using the simple gap method and then, upon failing,

evaluated using the push-pull method. Rather, the set of jobs to be inserted first pass through the job insertion heuristic using the faster simple gap implementation. If there are unscheduled jobs remaining after the first pass through the simple gap method, then the jobs are passed through the job insertion heuristic using the push-pull implementation.

Chapter 6 Application Interface

6.1 Interface Design Objectives

Prior to this work, TMC used an MS Excel/Matlab implementation of the algorithm developed by Bombien (2000) to generate their periodic route schedules. One of the objectives of this work is to present an implementation of the revised scheduling algorithms in a user-friendly Visual Basic 6.0 graphical user interface.

The primary design objectives for the interface are: 1) present an overall view of the input schedule; 2) present an overall view of the output route schedule generated; 3) support importing of old Excel format input schedules; and 4) allow simple, yet comprehensive job input and modification. Screenshots and short descriptions are presented in section 6.2 to show how these design objectives are met by the graphical user interface developed in this work.

6.2 Interface Screenshots

Input Schedule Viewer

Input Data - openmatlabmacro.xls*

Job	Destination	Depart Day	Depart Time	Delivery Day	Delivery Time	Return Day	Return Time	Miles	Time Window	Exclusion Period
1	Arlington, TX 76011	Wednesday	20:30	Thursday	07:00	Thursday	19:30	944	Anytime	None
2	Arlington, TX 76011	Wednesday	20:30	Thursday	07:00	Thursday	19:30	944	Anytime	None
3	Bethlehem, PA / Delran, NJ 08075 (Delran/Perryman)	Tuesday	13:00	Wednesday	15:00	Thursday	17:00	2098	None	None
4	Carol Stream, IL 60188	Thursday	16:30	Friday	05:00	Friday	19:30	1102	Anytime	None
5	Carol Stream, IL 60188	Thursday	16:30	Friday	05:00	Friday	19:30	1102	Anytime	None
6	Charlotte, NC /Shenandoah, GA 30265	Wednesday	12:00	Thursday	10:00	Thursday	20:30	1268	None	None
7	Crawfordsville, IN /New Castle, PA 16101	Tuesday	18:00	Wednesday	15:00	Thursday	10:30	1653	Anytime	None
8	Crawfordsville, IN 16101	Thursday	20:00	Friday	06:00	Friday	18:00	898	None	None
9	Dayville, CT 16241	Thursday	00:30	Friday	05:30	Saturday	12:30	2520	None	None
10	Dayville, CT 16241	Tuesday	00:30	Wednesday	05:30	Thursday	12:30	2460	None	None
11	Dayville, CT / Bohemia, NY 11716	Wednesday	01:30	Thursday	12:00	Friday	16:00	2607	None	None
12	Dayville, CT /Bohemia, NY 11716	Sunday	01:30	Monday	12:00	Tuesday	16:00	2607	None	None
13	Delran, NJ 08075	Tuesday	16:00	Wednesday	15:00	Thursday	16:00	2042	Anytime	None
14	Duluth, GA 30136	Wednesday	20:00	Thursday	05:30	Thursday	17:00	818	Anytime	None
15	Duluth, GA 30136	Wednesday	20:00	Thursday	05:30	Thursday	17:00	818	Anytime	None
16	Elk Grove Village, IL 60007	Sunday	08:30	Sunday	21:00	Monday	11:30	1096	None	None
17	Elk Grove Village, IL 60007	Monday	16:30	Tuesday	05:00	Tuesday	19:30	1096	None	None
18	Elk Grove Village, IL (Carol Stream, IL 60188)	Wednesday	16:30	Thursday	08:00	Thursday	22:30	1128	None	None
19	Elk Grove Village, IL /Windsor, WI 53598	Tuesday	16:30	Wednesday	10:00	Thursday	03:00	1334	None	None
20	Elk Grove Village, IL/Windsor, WI 53598	Thursday	16:30	Friday	10:00	Saturday	03:00	1334	None	None
21	Ft Lauderdale/Jupiter/Orlando, FL 32809	Thursday	06:00	Friday	14:00	Saturday	10:30	2054	None	None
22	Ft Lauderdale/Jupiter/Orlando, FL 32809	Tuesday	06:00	Wednesday	14:00	Thursday	10:30	2054	None	None
23	Ft Lauderdale (Lakeland, FL)	Sunday	06:00	Monday	05:00	Tuesday	07:30	2071	None	None
24	Ft Lauderdale (Lakeland, FL)	Wednesday	06:00	Thursday	12:00	Friday	06:30	2071	None	None
25	Garland, TX	Sunday	22:00	Monday	07:00	Monday	18:00	902	None	None
26	Garland, TX	Wednesday	22:00	Thursday	07:00	Thursday	18:00	902	None	None

Figure 6.1 Input Schedule Viewer with data imported from Excel file

The screenshot in Figure 6.1 shows a job input schedule imported from an existing input schedule in the MS Excel format. In this view, each row represents an input job, and all of the properties of a job can be observed, but not modified directly. In order to modify the data for a particular job, the user must double-click on the row pertaining to that job.

Job Input

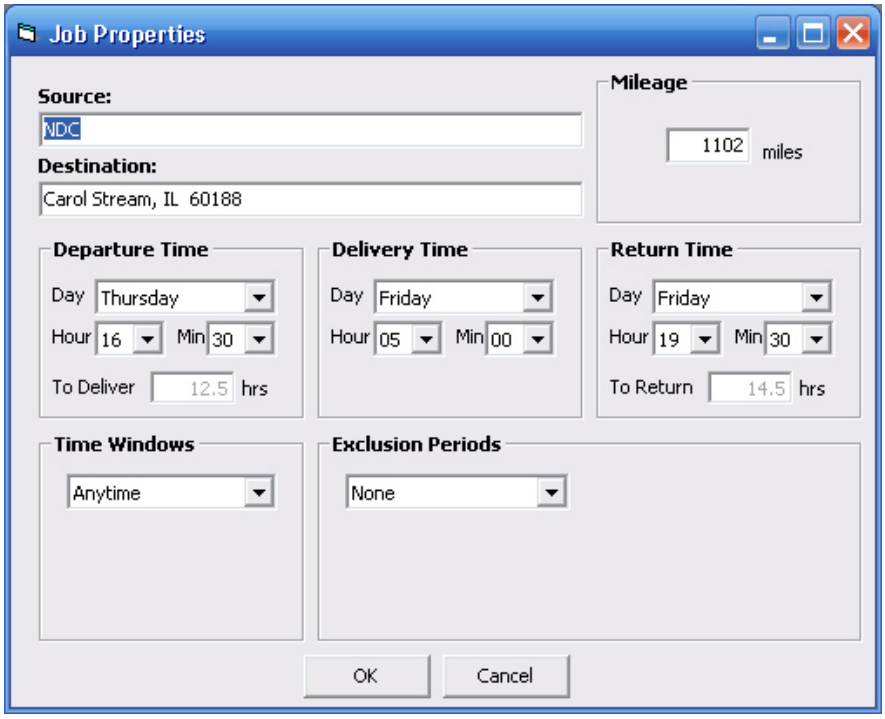


Figure 6.2 Job Input Dialog

When a user double-clicks on a job-row in the Input Schedule Viewer, a dialog that resembles the one in Figure 6.2 pops up. In this dialog, the user can alter the job's properties such as departure, delivery, and return times, applicable time windows, and job mileage, to name a few. When the user is done making changes, they click "Ok" to save the changes and or "Cancel" to discard the changes, and then return to the Input Schedule Viewer. Any changes saved will then be reflected on the Input Schedule Viewer. The user will also see a dialog box similar to the one shown in Figure 6.2 when inserting a new job into the input schedule. In this case, the dialog box will appear with blank fields.

Scheduler Run Properties

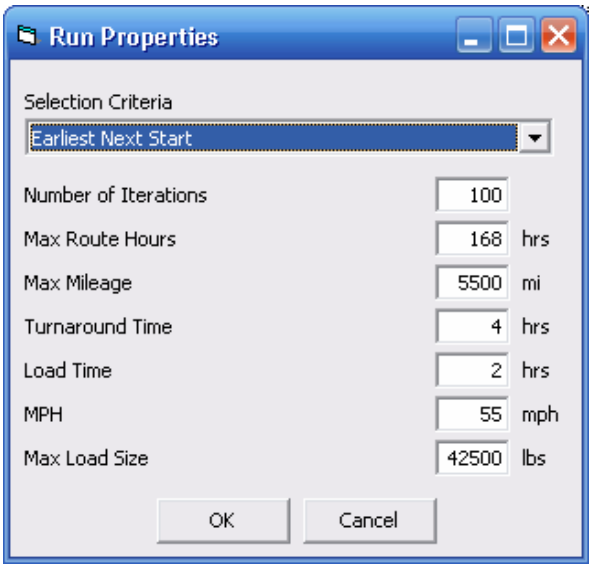
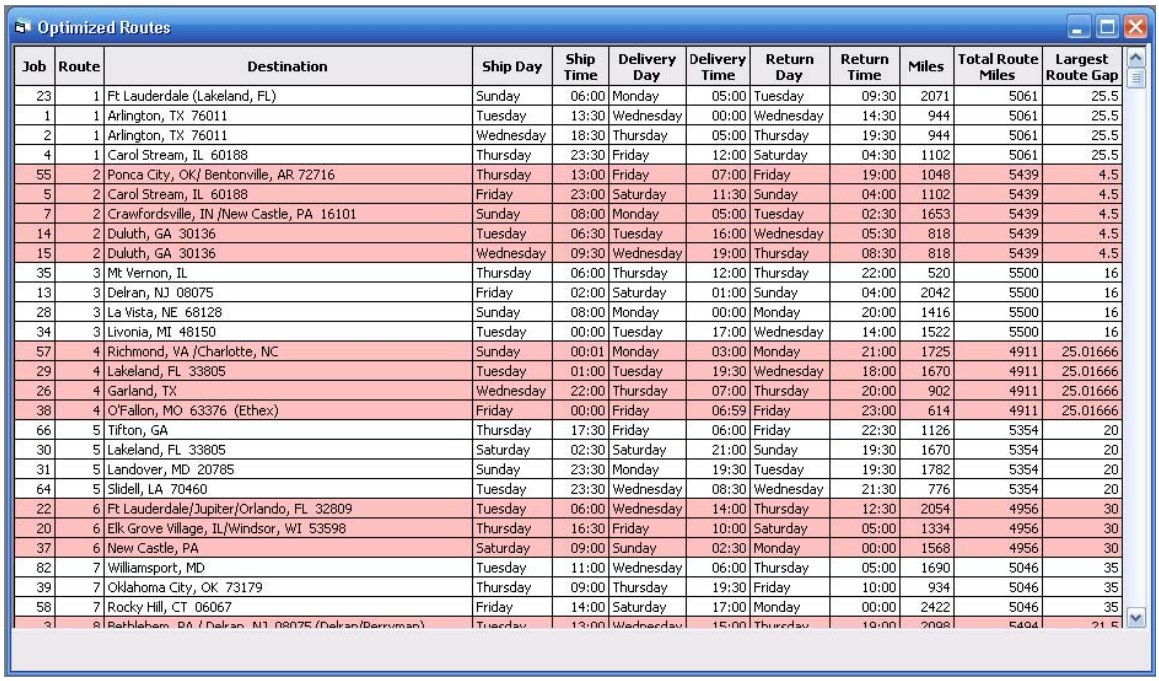


Figure 6.3 Run Properties dialog.

Once the user has input all of the jobs to be scheduled, they select a menu option to generate an optimized route schedule for the jobs. Selecting the "Generate Routes" menu option triggers the dialog box shown in Figure 6.3 to appear. With this dialog, the user can set the parameters used by the scheduling algorithm to generate routes, including the selection method to be used by the algorithm for constructing routes.

Output Route Schedule Viewer



Optimized Routes

Job	Route	Destination	Ship Day	Ship Time	Delivery Day	Delivery Time	Return Day	Return Time	Miles	Total Route Miles	Largest Route Gap
23	1	Ft Lauderdale (Lakeland, FL)	Sunday	06:00	Monday	05:00	Tuesday	09:30	2071	5061	25.5
1	1	Arlington, TX 76011	Tuesday	13:30	Wednesday	00:00	Wednesday	14:30	944	5061	25.5
2	1	Arlington, TX 76011	Wednesday	18:30	Thursday	05:00	Thursday	19:30	944	5061	25.5
4	1	Carol Stream, IL 60188	Thursday	23:30	Friday	12:00	Saturday	04:30	1102	5061	25.5
55	2	Ponca City, OK/ Bentonville, AR 72716	Thursday	13:00	Friday	07:00	Friday	19:00	1048	5439	4.5
5	2	Carol Stream, IL 60188	Friday	23:00	Saturday	11:30	Sunday	04:00	1102	5439	4.5
7	2	Crawfordsville, IN /New Castle, PA 16101	Sunday	08:00	Monday	05:00	Tuesday	02:30	1653	5439	4.5
14	2	Duluth, GA 30136	Tuesday	06:30	Tuesday	16:00	Wednesday	05:30	818	5439	4.5
15	2	Duluth, GA 30136	Wednesday	09:30	Wednesday	19:00	Thursday	08:30	818	5439	4.5
35	3	Mt Vernon, IL	Thursday	06:00	Thursday	12:00	Thursday	22:00	520	5500	16
13	3	Delran, NJ 08075	Friday	02:00	Saturday	01:00	Sunday	04:00	2042	5500	16
28	3	La Vista, NE 68128	Sunday	08:00	Monday	00:00	Monday	20:00	1416	5500	16
34	3	Livonia, MI 48150	Tuesday	00:00	Tuesday	17:00	Wednesday	14:00	1522	5500	16
57	4	Richmond, VA /Charlotte, NC	Sunday	00:01	Monday	03:00	Monday	21:00	1725	4911	25.01666
29	4	Lakeland, FL 33805	Tuesday	01:00	Tuesday	19:30	Wednesday	18:00	1670	4911	25.01666
26	4	Garland, TX	Wednesday	22:00	Thursday	07:00	Thursday	20:00	902	4911	25.01666
38	4	O'Fallon, MO 63376 (Ethex)	Friday	00:00	Friday	06:59	Friday	23:00	614	4911	25.01666
66	5	Tifton, GA	Thursday	17:30	Friday	06:00	Friday	22:30	1126	5354	20
30	5	Lakeland, FL 33805	Saturday	02:30	Saturday	21:00	Sunday	19:30	1670	5354	20
31	5	Landover, MD 20785	Sunday	23:30	Monday	19:30	Tuesday	19:30	1782	5354	20
64	5	Slidell, LA 70460	Tuesday	23:30	Wednesday	08:30	Wednesday	21:30	776	5354	20
22	6	Ft Lauderdale/Jupiter/Orlando, FL 32809	Tuesday	06:00	Wednesday	14:00	Thursday	12:30	2054	4956	30
20	6	Elk Grove Village, IL/Windsor, WI 53598	Thursday	16:30	Friday	10:00	Saturday	05:00	1334	4956	30
37	6	New Castle, PA	Saturday	09:00	Sunday	02:30	Monday	00:00	1568	4956	30
82	7	Williamsport, MD	Tuesday	11:00	Wednesday	06:00	Thursday	05:00	1690	5046	35
39	7	Oklahoma City, OK 73179	Thursday	09:00	Thursday	19:30	Friday	10:00	934	5046	35
58	7	Rocky Hill, CT 06067	Friday	14:00	Saturday	17:00	Monday	00:00	2422	5046	35
3	8	Bethlehem, PA / Delran, NJ 08075 (Delran/Perryman)	Tuesday	13:00	Wednesday	15:00	Thursday	19:00	2098	5494	21.5

Figure 6.4 Generated Route Schedule Viewer.

The results of the scheduling algorithm are shown in a window like the one in Figure 6.4. At this point, the user can choose to save the generated route or discard it and generate a new one using different parameters. Each row represents an individual leg on a route, and the routes are distinguished by alternating colors as well as a column that contains the route number.

Chapter 7 Computational Experiments

7.1 Basic Scheduling Test Problems

In order to test the effectiveness of the basic route scheduling algorithms we developed a routine to generate a set of random problems. The location characteristics of the random delivery jobs are based on real FDC location data provided by TMC. Random problems are generated using the following routine.

1. Define a total number of delivery jobs, n, and a maximum number of jobs per FDC per week, *maxFDCJobs*.
2. Create an array *fdcJob* of size M, the number of FDC's
3. If $n > 0$, generate a random number between 0 and $M - 1$. When n=0, skip to step 5.
4. Increment *fdcJob*[M] by 1 if *fdcJob*[M] < *maxFDCJobs*, decrement n by 1, and return to Step 3. If *fdcJob*[M] ≥ 5, then do nothing and return to Step 3.
5. For all M FDCs, generate *fdcJob*[M] delivery jobs. For each job:
    a. Randomly generate a normal start time.
    b. Randomly generate a time window width (ww) between 0 and 24.
    c. Earliest start time is normal start time – ww/2. Latest start time is normal start time + ww/2.

7.2 Backhaul Scheduling Test Problems

To evaluate the backhaul carrying capacities of the solution routes generated for the basic scheduling problems, we randomly generated a set of backhaul job test problems. The backhaul jobs are assumed to be jobs that can be scheduled anytime, and their drop off location is assumed to be negligibly close to the NDC. The data associated with a single backhaul job problem is a location identification number and a load size. Mileage data is stored in arrays accessible by the location identification number. The load size for all problems generated is the system defined maximum load size that can be carried by a single trailer. The number of backhaul jobs per backhaul site location for each test problem is generated in a similar manner to that described above for the primary test problems.

7.3 Experiments

The scheduling heuristics described in this work were implemented in C++. Refer to Appendix A for the detailed code. To evaluate the heuristics at different levels, six batches of 1,000 problems were generated for each of six job sizes: 20, 40, 60, 80, 100, and 150 jobs. For each basic scheduling problem, corresponding backhaul job scenarios of sizes 5, 10, 20, 40, 80, and 100 jobs were created. An experimental run for a problem consisted of 1,000 randomized iterations of each of the four scheduling methods. During each iteration, the backhaul scheduling routine evaluated the backhaul carrying capabilities of the solution route constructed using each of the six backhaul job scenarios associated with the given problem. Raw data was collected from each iteration and includes the number of routes constructed and the number of backhauls carried by the route schedule for each backhaul scenario.

7.4 Lower Bounds

Comparison of the route schedule solutions with possible lower bounds of the test problems provides insight into the quality of the heuristics. In this work, three lower bounds are used. The first two are derived from a bin-packing formulation of our stated problem. The third lower bound examines the delivery time window characteristics of the scheduling problem. The two bin-packing lower bounding schemes focus on the system defined constraints of maximum weekly mileage and maximum weekly processing time. The processing time for a job is not always directly proportional to the mileage associated with the job. Therefore, the two constraints must be observed separately.

First, for the bin packing formulation, consider the given system constraint with a maximum size, b, as the bin size. Each job has a value c_i associated with this constraint, and for every job, $c_i < b$ must be true. From this, we determine the minimum number of bins, n, required for N jobs by the expression

$$n = \frac{\sum_{i=1}^{N} c_i}{b}$$

The bin packing formulation is used to calculate the lower bound derived from the maximum weekly mileage constraint and the lower bound derived from the maximum weekly processing time constraint.

The third lower bound procedure examines the input delivery schedule and determines the minimum number of trucks necessary to complete the deliveries based on job overlap. From the time window properties, we know that a truck must be in use for a particular job *j* between the job's latest departure time, b_j, and its earliest completion time, c_j. By iterating through each job in the input schedule the number of simultaneously active tractors (SAT) can be determined for each hour of the week. Over the 168 hours of the week, the maximum SAT value represents the minimum number of tractors required in the fleet to complete the input job schedule.

7.5 Results

The following graphs show the performance of each heuristic versus the best lower bound for each problem. The sets of graphs also include a comparison between the best solution over all four selection methods versus the best lower bound for each problem. Backhaul carrying performance is not evaluated in this portion of the results.

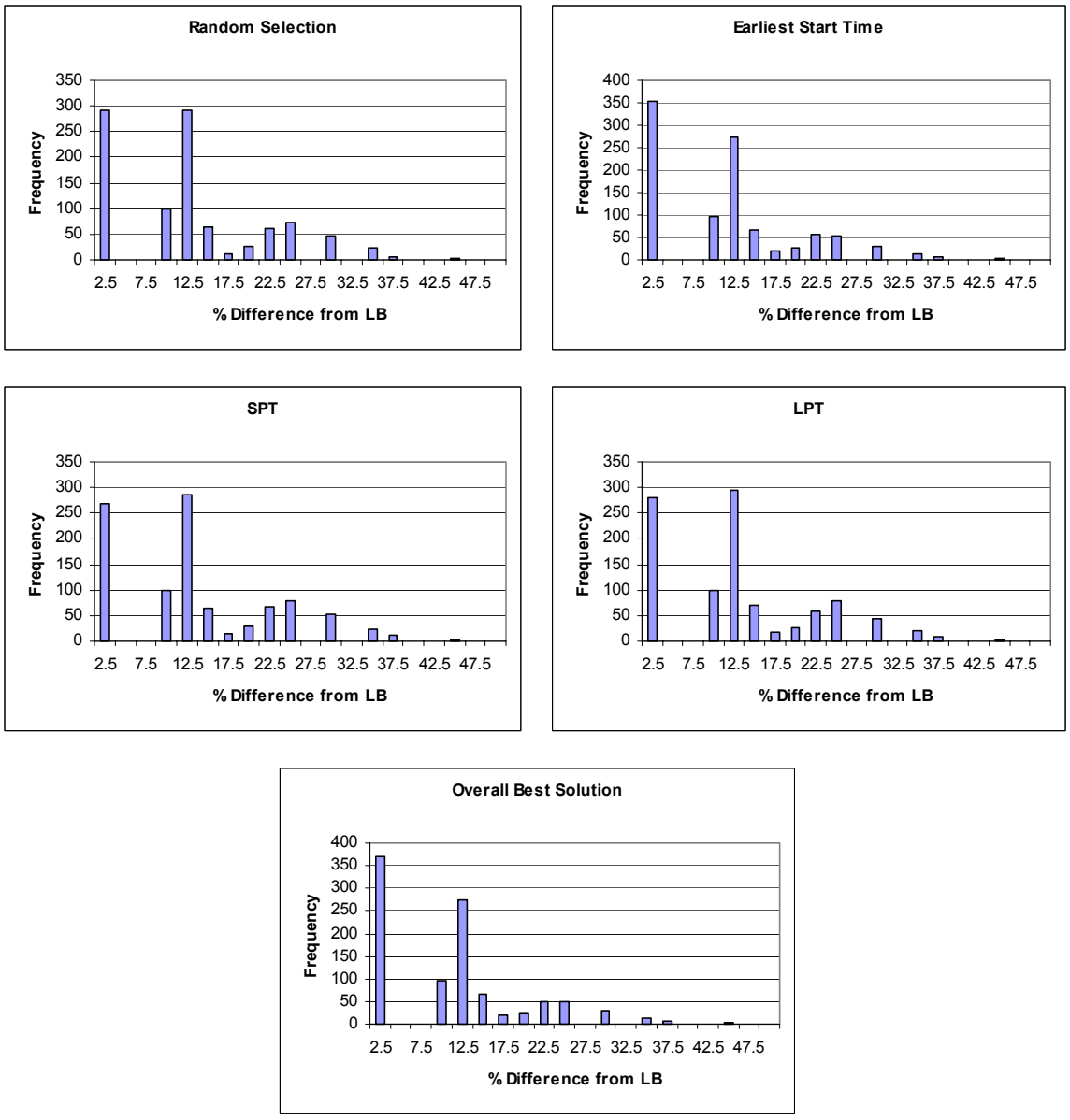


Figure 7.1 Histogram of selection method performance vs. best lower bound (20 Jobs)

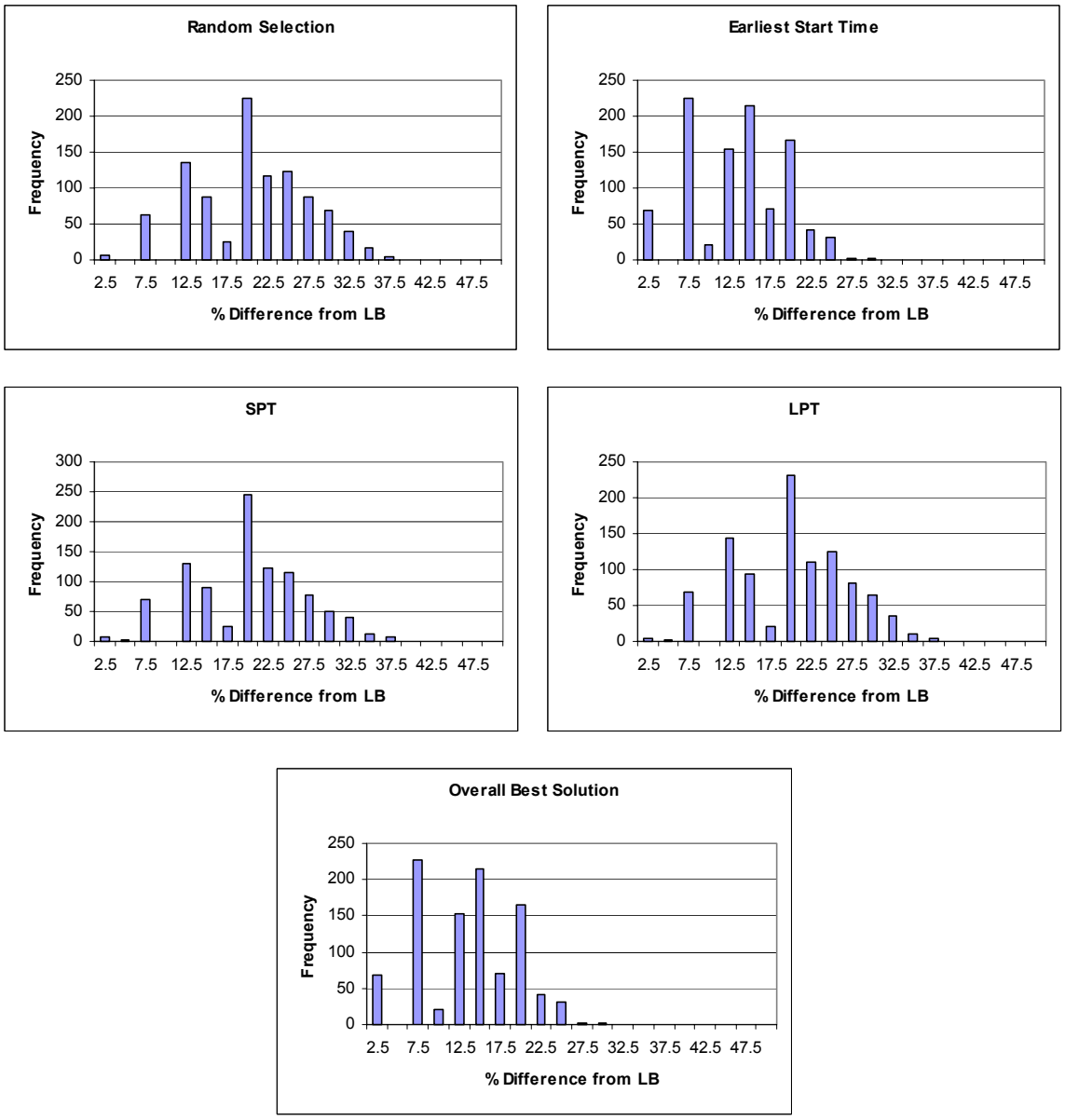


Figure 7.2 Histogram of selection method performance vs. best lower bound (40 Jobs)

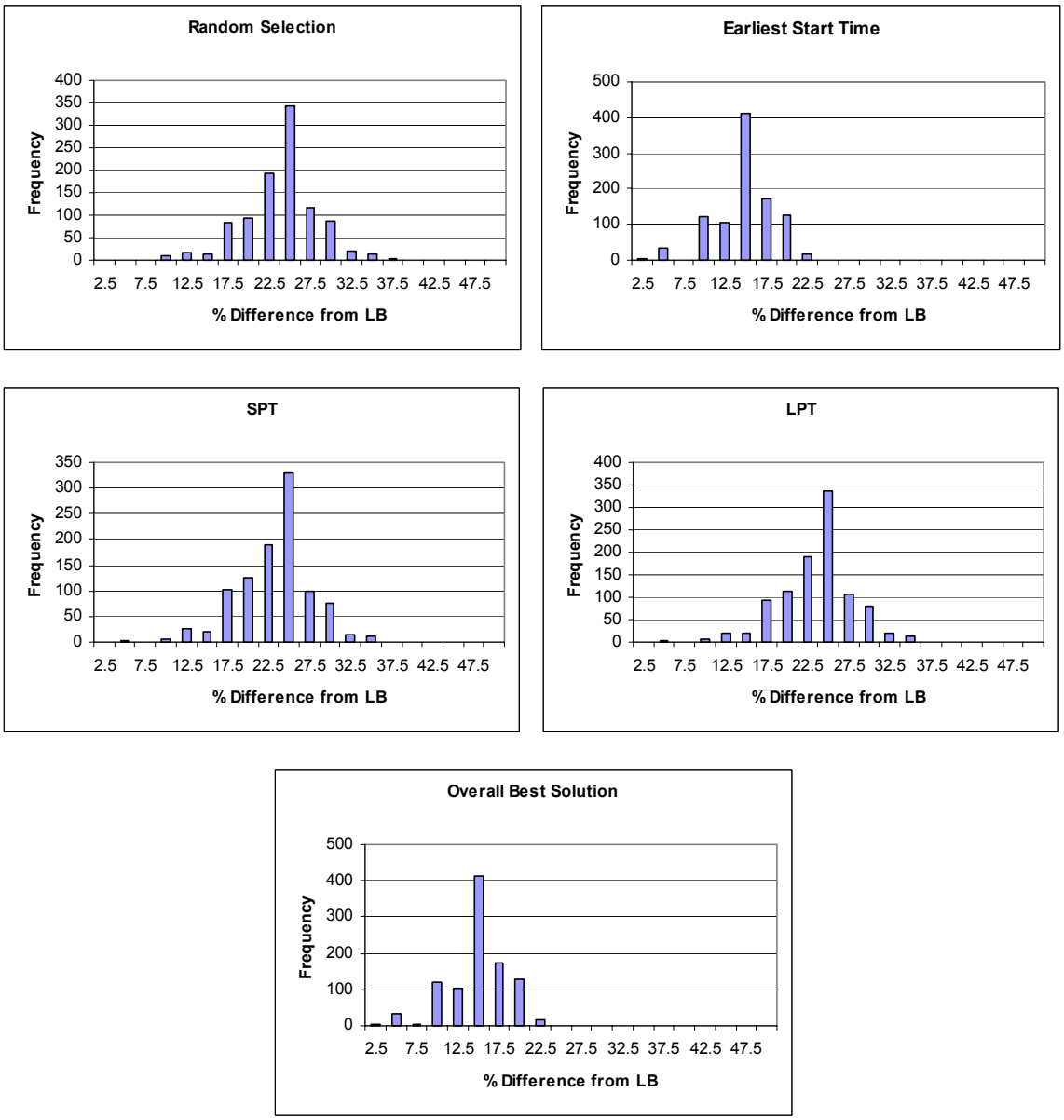


Figure 7.3 Histogram of selection method performance vs. best lower bound (60 Jobs)

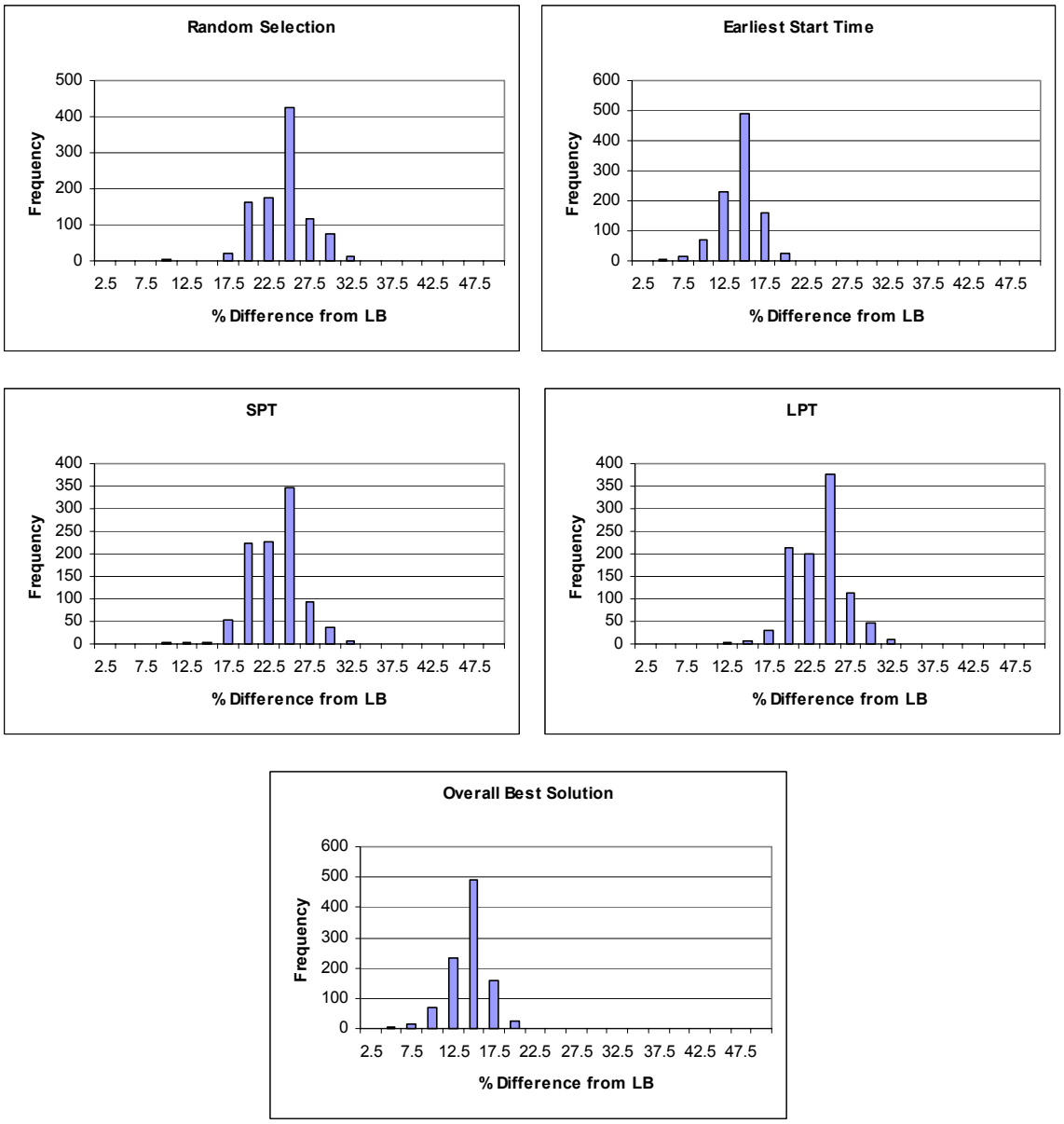


Figure 7.4 Histogram of selection method performance vs. best lower bound (80 Jobs)

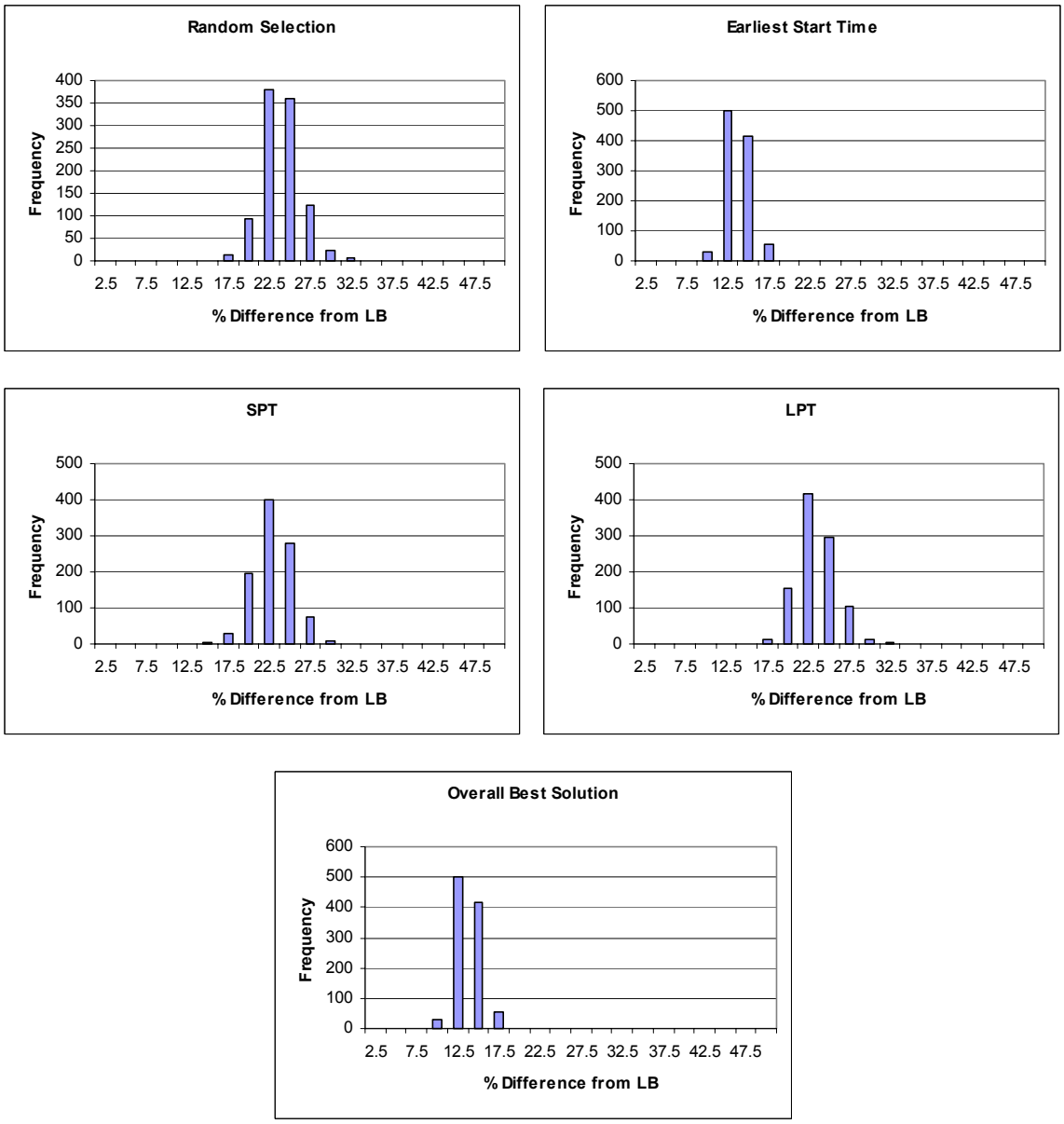


Figure 7.5 Histogram of selection method performance vs. best lower bound (100 Jobs)

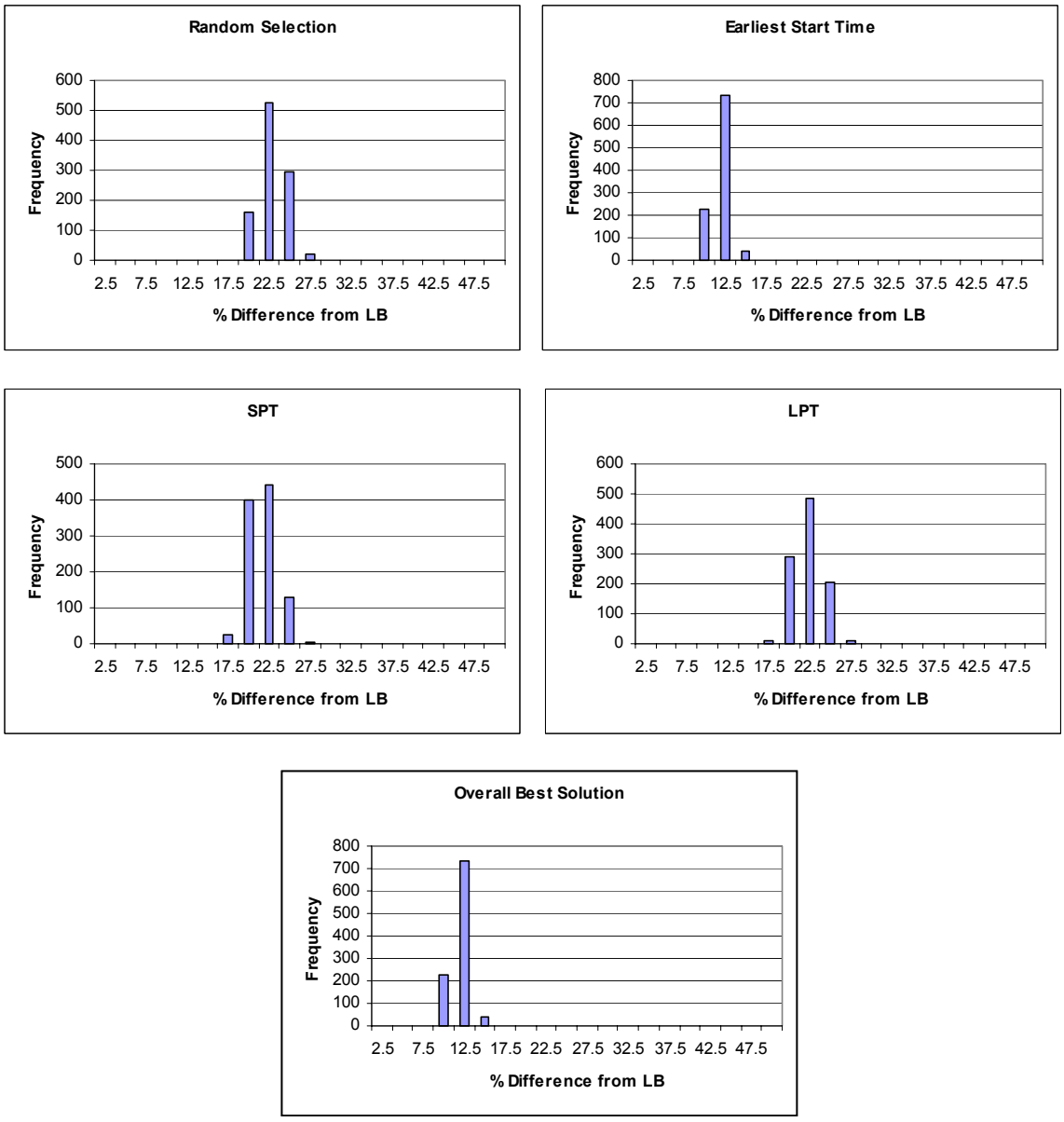


Figure 7.6 Histogram of selection method performance vs. best lower bound (150 Jobs)

	LPT			SPT			EST			Rand		
Problem Size	% Diff from best LB	% Diff St. Dev	Avg. CPU Time (sec)	% Diff from best LB	% Diff St. Dev	Avg. CPU Time (sec)	% Diff from best LB	% Diff St. Dev	Avg. CPU Time (sec)	% Diff from best LB	% Diff St. Dev	Avg. CPU Time (sec)
20	11.89	0.92	0.14	12.42	0.98	0.14	10.19	0.87	0.14	11.74	0.96	0.13
40	19.27	0.48	0.44	19.23	0.49	0.43	12.51	0.37	0.45	19.64	0.50	0.40
60	22.51	0.20	0.89	22.26	0.19	0.88	13.99	0.13	0.91	22.90	0.20	0.80
80	22.64	0.10	1.56	22.26	0.10	1.54	13.23	0.06	1.59	23.17	0.10	1.34
100	22.38	0.06	2.40	21.93	0.06	2.38	12.43	0.03	2.46	22.91	0.06	2.09
150	21.15	0.03	5.50	20.64	0.03	5.42	10.75	0.01	5.60	21.84	0.03	4.37

Table 7.1: Selection method performance

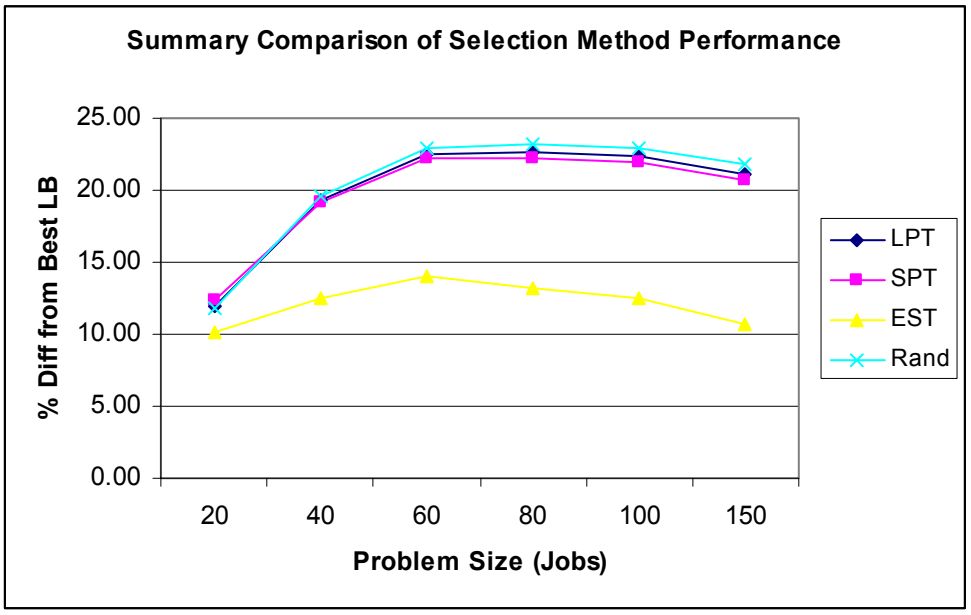


Figure 7.7 Summary selection method performance comparison

The histograms (Figures 7.1-7.6) and the performance summary (Figure 7.7) show that the Earliest Start Time (EST) selection method performs much better than the other three, especially as the number of jobs increases. The performance of the other three selection methods (LPT, SPT, and Random) is generally worse than EST, but similar to each other. The EST method delivers solutions that range in difference from the best lower bound from 10% to 15%. The other solutions generally range from 10% to 20%. For the 20 job size problems, a 12% difference between the lower bound and the solution generated equates to about one route. Similar conversions can be made for the remaing solution sets: 40 jobs – 20% difference equates to approximately three routes; 60 jobs – 14%

difference equates to approximately three routes; 80 jobs – 13% difference equates to approximately 4 routes; 100 jobs – 12% equates to approximately 4 routes; 150 jobs – 11% equates to approximately 6 routes. It is also important to note that all selection methods deliver solutions that do not differ from the best lower bound by more than 50%.

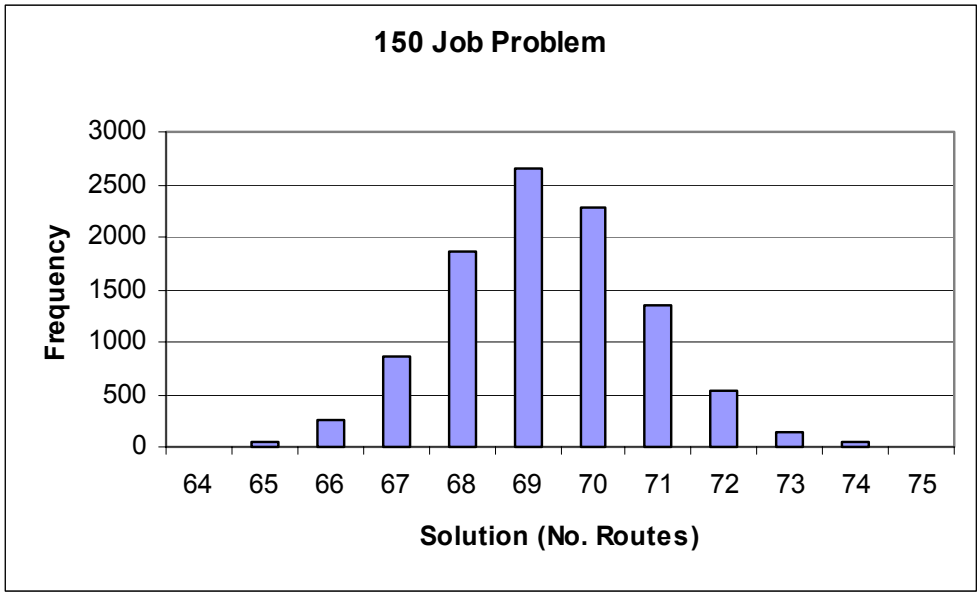


Figure 7.8 Histogram for 10,000 iterations of Random selection method

For the 20-job and 40-job size problems, the best lower bound was generally the SAT calculation. For the 60-job size problems, the three lower bounds generally provided similar results, and from the 80-job size problems up to the 150-job size problems, the mileage constraint bin-packing lower bound provided the best lower bound. The histogram in Figure 7.8 depicts the performance of the Random selection method over 10,000 iterations for a randomly selected problem from the 150 job problem set. The best lower bound from this problem suggests an optimal solution of 52 routes. As the histogram in Figure 7.8 shows, the solutions generated by the Random selection method are approximately normally distributed around a mean of about 69 routes, approximately a 33% difference. This result demonstrates that, as in previous research, our analysis is plagued by weak lower bounds.

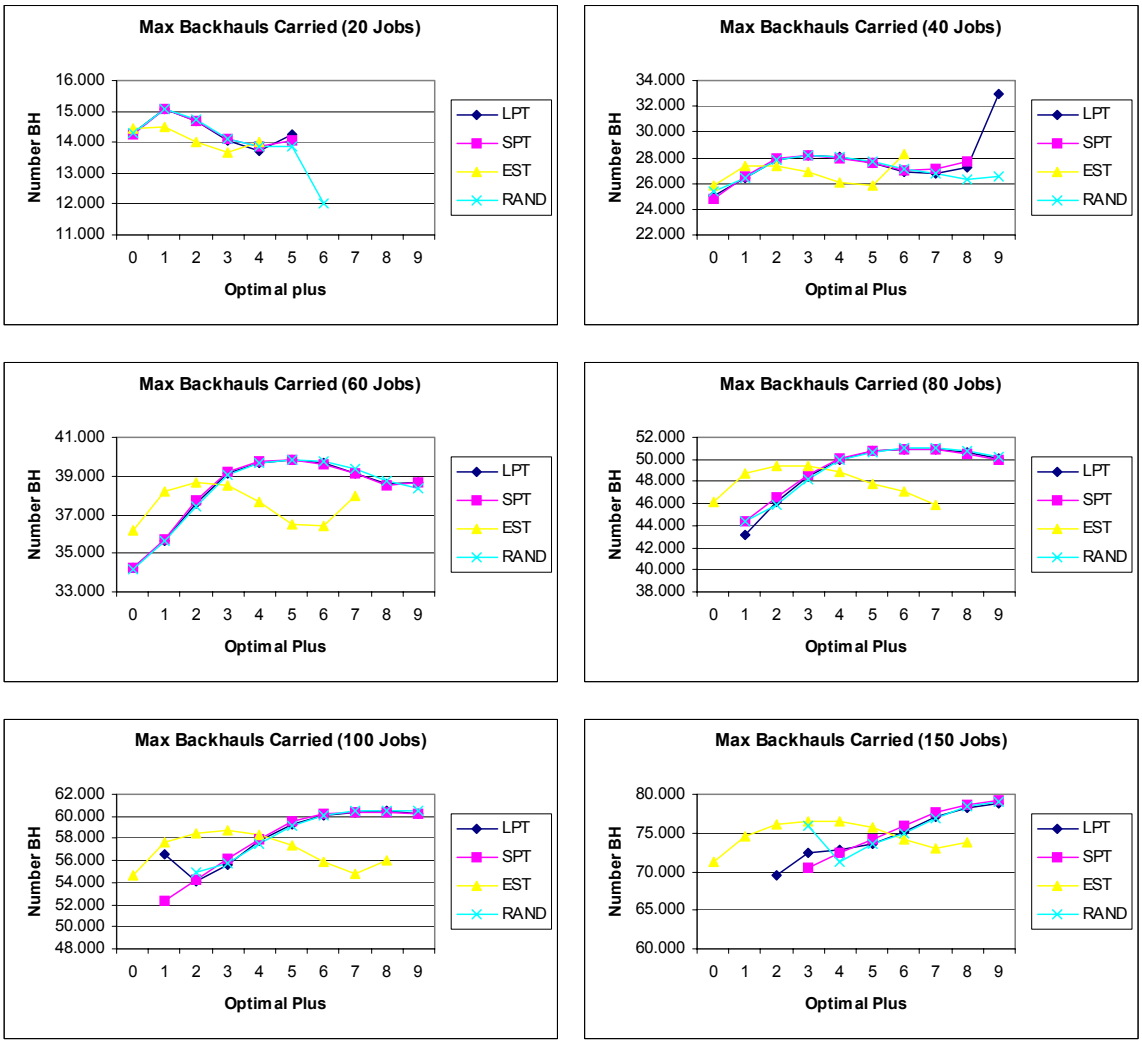


Figure 7.9 Maximum backhauls carried at each of 10 optimality levels

In order to evaluate the flexibility for inserting backhauls into the route schedules generated, we tracked the number of backhauls carried by the solution schedule generated at each of the 1000 iterations. Then, we obtained the maximum number of backhauls carried when the solution schedule at each iteration yielded: a) the optimal number of routes, x, (best solution of the four selection methods), and then b) $x + k$, for $k = 1, 2, 3 \ldots 9$. We then averaged these values over the 1000 test problems. This analysis provides insight into the value of adding routes and tractors versus added backhaul carrying capabilities.

As shown in Figure 7.9, the EST selection method provides the greatest opportunity for increasing backhaul carrying capabilities by adding more tractors up until

the addition of about four tractors. While the ultimate decision with regards to the number of tractors to add depends on a cost-benefit analysis, we will assume that the cost of an additional tractor is significantly more than the added benefit of one more backhaul carry. Based on this assumption and the results shown in Figure 7.9, the EST selection method may offer increased backhaul carrying opportunities by adding one tractor, but the value of adding more than one tractor depreciates rapidly.

An unexpected result shown by Figure 7.9 is a decreasing tendency in the maximum backhauls carried as the number of tractors in use increases for the EST selection method. Presumably, adding more tractors would lead to an increase in the number of backhauls carried. It is important to remember, though, that increasing the number of tractors does not directly increase the number of opportunities for carrying backhauls. The increase in backhaul carrying opportunities is expected because it is assumed that solutions with more routes have longer idle times between legs. The longer idle times, then, translate into greater flexibility for scheduling backhauls.

While this particular property of the solution schedules holds true for the solutions generated by the LPT, SPT, and Random selection methods, those solutions generated using the EST selection method exhibit a different behavior. The primary purpose of the EST selection method is to reduce the idle time between legs. Even if a particular solution generated by EST is slightly greater than the optimal solution, the routes in the solution still possess the same reduced idle time characteristics. The extra routes, then, generally only have one or two jobs assigned to them, and the added flexibility for carrying backhauls is minimal.

Because the effect of additional tractors in the EST generated schedules on the backhaul carrying flexibility is minimal, randomization factors more heavily in the backhaul carrying performance of the EST solution schedules. Specifically, the performance is closely related to the order in which jobs are scheduled on route legs and the location of the backhaul jobs to be scheduled. In other words, the results shown in the graphs in Figure 7.9 for the EST method may be influenced more by the particular combinations of solution schedules and test problems than by the actual performance of the algorithm in place.

	Average CPU Time (per problem)							
	LPT		SPT		EST		Rand	
Problem Size	No Backhauls	With Backhauls	No Backhauls	With Backhauls	No Backhauls	With Backhauls	No Backhauls	With Backhauls
20	0.14	0.75	0.14	0.78	0.14	0.76	0.13	0.74
40	0.44	1.09	0.43	1.09	0.45	1.10	0.40	1.05
60	0.89	1.67	0.88	1.66	0.91	1.71	0.80	1.57
80	1.56	2.43	1.54	2.42	1.59	2.50	1.34	2.27
100	2.40	3.39	2.38	3.37	2.46	3.50	2.09	3.01
150	5.50	6.82	5.42	6.84	5.60	7.02	4.37	5.67

Table 7.2: CPU time for each selection method

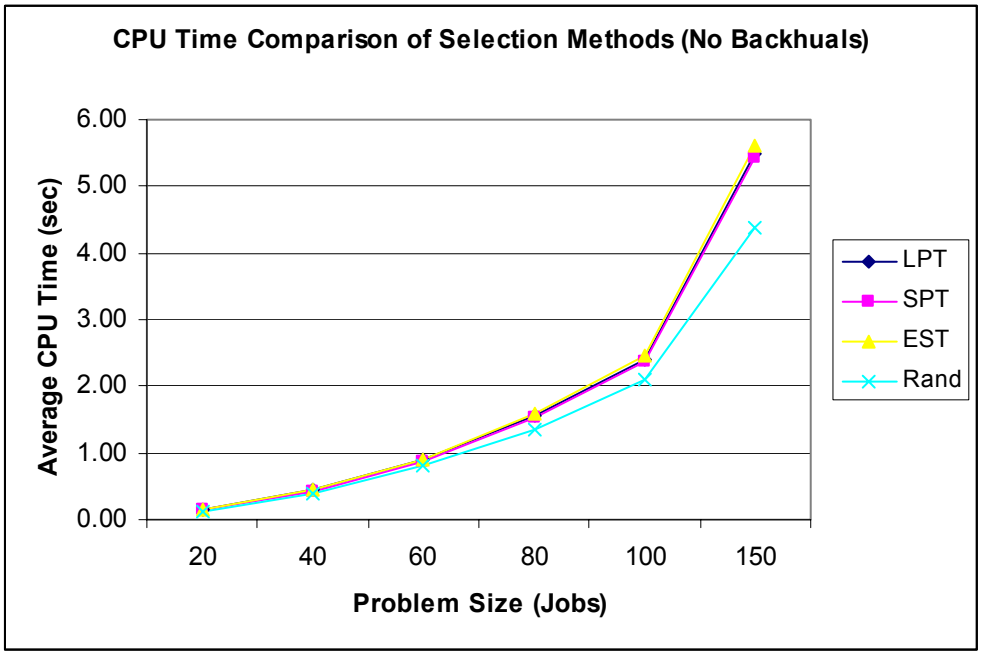


Figure 7.10: CPU Time with no backhaul scheduling

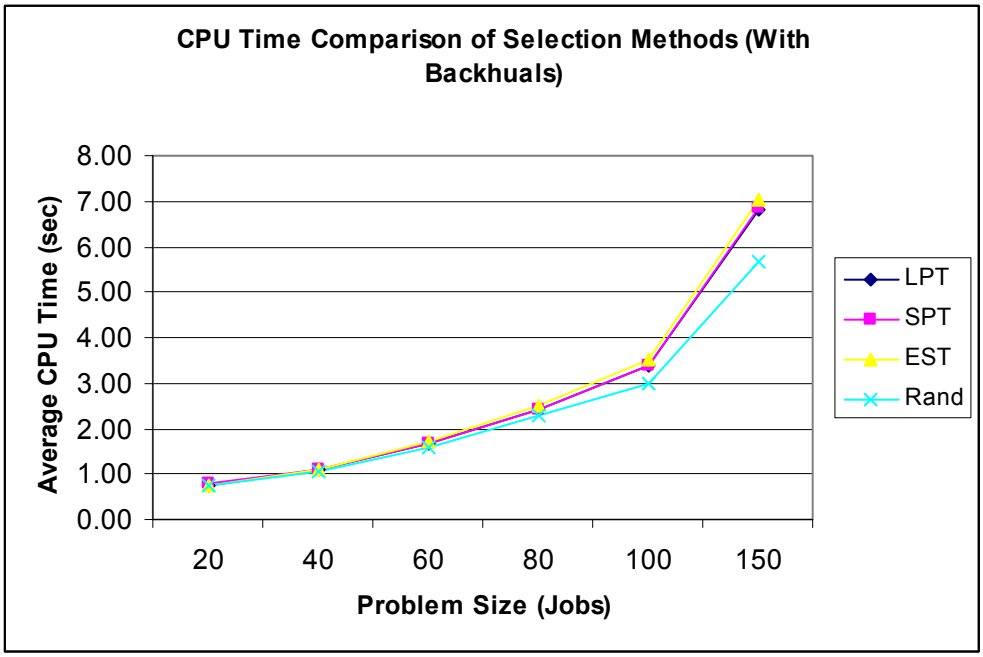


Figure 7.11: CPU Time with backhaul scheduling

The results in Figures 7.10 and 7.11 show that the Random selection method executes slightly faster than the other methods for larger problems. The Random selection method is the only method that does not require a sort function call when choosing the next job.

Chapter 8 Conclusions and Future Work

8.1 Conclusions

In this research we examined a single demand-point vehicle routing problem with the objective of minimizing the number of tractors required to service the customer. Our problem differs from traditional vehicle routing problems in that only one delivery is made per trip leading to a fixed total travel mileage over all jobs. Thus, our objective is to minimize the number of tractors in use rather than the overall miles traveled. As an important secondary focus, we investigated opportunities for carrying backhauls. For the basic scheduling problem, three lower bounding procedures were used: two of them addressed the problem as a bin-packing problem with maximum route constraints as the bin size; the third computed a lower bound based on the number of simultaneously active vehicles necessary to complete the job schedule based on the delivery time windows.

Because the work by Labiad (2000) showed that the Forward heuristic generally provided the best results, this work focused on ways to improve the performance of the Forward heuristic. To enhance the Forward heuristic, we formulated four selection criteria for constructing the routes and applied some modifications to better identify candidate routes for selection and pack the routes more efficiently. All of the selection criteria provided reasonable solutions. However, the Earliest Start Time selection criteria yielded the best results, differing by an average of no more than 15% from the best lower bound in large problems. The EST selection method also provided the most flexibility for carrying backhauls when tested at or in close proximity to the optimum value obtained for a given problem.

In addition to the general scheduling heuristic, we introduced a couple of job insertion heuristics to handle non-recurring jobs, schedule disruptions, or for solution improvement. The job insertion heuristics were not evaluated in our experiments but executed as expected.

8.2 Future Work

There are several areas that can be explored as an extension of this work:

- Another problem TMC faces is driver turnover, which can be aggravated by having routes that are unattractive (i.e. lacking significant downtime) to drivers.

Decreasing the efficiency of routes contradicts the objective of the heuristic in this work, however, the work load can be balanced by combining the truck scheduling with the crew scheduling problem. Some research for a simultaneous vehicle and crew scheduling problem exists for the airline industry and public transportation (Freling et al, 2003; Haase et al, 2001; Klabjan et al, 1999), but not for our specific problem.

- There may be some opportunity for savings by exploring possible schedule improvement routines. These routines could analyze the output schedules and search for gaps in the schedule. The job insertion methods discussed in this work may potentially be used for improvement, but there may also be other routines.
- Improved backhaul scheduling algorithms may present more opportunities to carry backhauls. Discussions of improved backhaul scheduling algorithms may also require refinements in the definition of the backhaul job problem and the metrics used to evaluate the effectiveness of solution methods.

Bibliography

Bodin, L., A. Mingozzi, R. Baldacci, M. Ball (2000) "The Rollon-Rolloff Vehicle Routing Problem," *Transportation Science*, 34(3), 271-288.

Bombien, R (2001) "Generating Scheduled Routes for Trucking Services".

Desrochers, M., J. K. Lenstra, M. W. P. Savelsbergh, and F. Soumis (1988) "Vehicle Routing with Time Windows: Optimization and Approximation," In *Vehicle Routing: Methods and Studies* by B. L. Golden and A. A. Assad (eds.). North Holland, Amsterdam, 65-84.

Desrochers, M., J. Desrosiers, and M. Solomon (1992) "A New Optimization Algorithm for the Vehicle Routing Problem with Time Windows," *Operations Research*, 40(2), 342-354.

Fischetti, M., S. Martello, P. Toth (1989) "The Fixed Job Schedule Problem with Working-Time Constraints," *Operations Research*, 37(3), 395-403.

Fisher, M. L., K. O. Jornsten, O. B. G. Madsen (1997) "Vehicle Routing with Time Windows: Two Optimization Algorithms," *Operations Research* 45(3), 488-492.

Freling, R., D. Huisman, A. P. M. Wagelmans (2003) "Models and Algorithms for Integration of Vehicle and Crew Scheduling," *Journal of Scheduling,* 6, 63-85.

Haase, K., G. Desaulniers, and J. Desrosiers (2001) "Simultaneous Vehicle and Crew Scheduling in Urban Mass Transit Systems," *Transportation Science,* 35(3), 286-303.

Klabjan, D., E. L. Johnson, and G. L. Nemhauser (1999) "Airline Crew Scheduling with Time Windows and Plane Count Constraints".

Kohl, N. and O. B. G. Madsen (1997) "An Optimization Algorithm for the Vehicle Routing Problem with Time Windows Based On Lagrangian Relaxation," *Operations Research*, 45(3), 395-406.

Labiad, N. (2002) "Scheduling and Routing of Vehicles for a Transportation Company".

Solomon, M. M. (1987) "Algorithms for Vehicle Routing and Scheduling Problems with Time Window Constraints," *Operations Research*, 35, 254-265.

Solomon, M. M., and J. Desrosiers (1988) "Time Window Constrained Routing and Scheduling Problems," *Transportation Science*, 22, 1-13.

Appendix A – (C++ Code)

csched.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <ctime>
#include "jobtypes.h"
#include "arrayfuncs.h"
#include "gfuncs.h"

#define RAND_SEED 1000

using namespace std;

void fixedBHLite(struct inp_sched *myInpSched, struct run_settings
*myRunSetup, struct bh_data *myBHData, int pnum, char filename[]);
void fixedNoBH(struct inp_sched *myInpSched, struct route_sched
*myRouteSched, struct run_settings *myRunSetup);
int packBackHauls(struct route_sched *packRoutes, backhaul schedBH[],
int numBH, struct run_settings *myRunSetup);
void pushRoute(route *clayRoute, int frzJobIdx, float rteEndTime, float
rteStartTime, int debugrteIdx);
int pushPullGapCheck(route *checkRoute, int rteIdx, job *newJob, float
newJobPi);
int simpleGapCheck(route *checkRoute,int rteIdx, job *newJob, float
newJobPi);
int simpleGapInsert(route *jamRoute,int afterJobIdx,job *newGuy,float
newGuyStart, float newGuyPi,float loadTime);
void insertJobs(struct inp_sched *myInpSched, struct route_sched
*myRouteSched, struct run_settings *myRunSetup);
int threesCompany(int idxNewJob,struct inp_sched *myInpSched, struct
route_sched *myRouteSched,struct run_settings *myRunSetup);
int rollDice(int rdmUpBnd);
int jobOnLB(job schedJobs[], int numSchedJobs, struct run_settings
*myRunSetup);
int milesLB(job schedJobs[], int numSchedJobs, int maxMiles);
int hoursLB(job schedJobs[], int numSchedJobs, struct run_settings
*myRunSettings);
float getLegBHLoad(route_job *gllRouteJob);
float getLegMileage(route_job *glmRouteJob);
float getRouteMileageByLeg(route *grmRoute);
float getRouteMileage(route *grmRoute);
float getLegPHours(route_job *glpRouteJob);
float getRouteProcessHours(route *grpRoute);
float calcTravelTime(int dist, int speed);
float calculateAdjustedStart(int rteIndex, int ji, float sngCki, job
*jobProps, float sngPi, float sngRteStart);
float calculateAdjustedPullStart(float sngCki, job *jobProps, float
sngPi, float sngRteStart);
float calculateAdjustedPushStart(float sngSkj, job *jobProps, float
sngPi, float sngRteStart);
void addJobToRoute(struct job *nJob,float nJobPi, int nJobIdx, float
nJobStart, struct route *cRoute, int routeLeg, struct run_settings
*myRunSetup);
```

```
int chooseNextJobByPT(int ptMode, int arrUfeas[], float arrPi[], int
ptNumUfeas);
int chooseNextJobByStart(int stMode, int arrUfeas[], int stNumUfeas,
float arrESTimes[]);
int chooseNextJobByRandom(int arrUfeas[], int rNumUfeas);
void grabInput(int inpType, inp_sched *myInpSched, char filename[]);
void grabBHProblem(struct bh_sched *myBHInpSched, char filename[]);
void grabBHMileData(struct bh_data *myBHData);
void displayInput(inp_sched *myInpSched);
void vomitRoutes(route_sched *myRouteSched);
void vomitRoutesAndBH(route_sched *myRouteSched);
void vomitRoutesAndInputSched(route_sched *myRouteSched, inp_sched
*myInpSched,char filename[]);
float getSolnRouteData(route_sched *myRouteSched,int data_type);


void grabInput(int inpType, inp_sched *myInpSched, char filename[]){
   ifstream in;
   char trash[5];
      int n;

      in.open(filename);
      if(!in){
            cout << "Cannot open input file.\n";
      }

      n=0;
      while(!in.eof()){
            if(inpType==0){
                  in >> myInpSched->inp_jobs[n].Mj >> trash >>
myInpSched->inp_jobs[n].chng_ind >> trash >> myInpSched->inp_jobs[n].Ej
>> trash >> myInpSched->inp_jobs[n].Dj >> trash >> myInpSched-
>inp_jobs[n].Lj >> trash >> myInpSched->inp_jobs[n].Cj >> trash >>
myInpSched->inp_jobs[n].Rj >> trash >> myInpSched->inp_jobs[n].ex_ind
>> trash >> myInpSched->inp_jobs[n].ExAj >> trash >> myInpSched-
>inp_jobs[n].ExBj;
            }
            else{
                  in >> myInpSched->inp_jobs[n].fdcIdx >> trash >>
myInpSched->inp_jobs[n].Mj >> trash >> myInpSched->inp_jobs[n].chng_ind
>> trash >> myInpSched->inp_jobs[n].Ej >> trash >> myInpSched-
>inp_jobs[n].Dj >> trash >> myInpSched->inp_jobs[n].Lj >> trash >>
myInpSched->inp_jobs[n].Cj >> trash >> myInpSched->inp_jobs[n].Rj >>
trash >> myInpSched->inp_jobs[n].ex_ind >> trash >> myInpSched-
>inp_jobs[n].ExAj >> trash >> myInpSched->inp_jobs[n].ExBj;
            }
            myInpSched->inp_jobs[n].resched_ind = 0;
            myInpSched->inp_jobs[n].idx = ++n;
      }

      myInpSched->numSchedJobs = n-1;

      in.close();
}
void grabBHProblem(struct bh_sched *myBHInpSched, char filename[]){
      ifstream in;
      char trash[5];
```

```
	int n;

	in.open(filename);
	if(!in){
		cout << "Cannot open backhaul problem file.\n";
	}

	n=0;
	while(!in.eof()){
		in >> myBHInpSched->inp_bh[n].locID >> trash >>
myBHInpSched->inp_bh[n].loadSize;
		myBHInpSched->inp_bh[n].bhID = ++n;
	}

	myBHInpSched->numBH = n-1;
	in.close();
}
void grabBHMileData(struct bh_data *myBHData){
	ifstream in,in2,in3;
	char filename[80];
	int n,i;

	//Grab Backhaul to Backhaul distances
	sprintf(filename,"ext/BH2BH.txt");
	in.open(filename);
	if(!in){
		cout << "Cannot open file.\n";
	}

	n=0;
	while(!in.eof()){
		for(i=0;i<NUM_BH_SITES;i++){
			in >> myBHData->BH2BH[n][i];
		}
		n++;
	}
	in.close();

	//Now Backhaul to NDC distance
	sprintf(filename,"ext/BH2NDC.txt");
	in2.open(filename);
	if(!in2){
		cout << "Cannot open BH2NDC.txt file.\n";
	}

	n=0;
	while(!in2.eof()){
		in2 >> myBHData->BH2NDC[n];
		n++;
	}
	in2.close();

	//And finally, Backhaul to FDC distances
	sprintf(filename,"ext/FDC2BH.txt");
	in3.open(filename);
	if(!in3){
		cout << "Cannot open FDC2BH.txt file.\n";
```

```
	}

	n=0;
	while(!in3.eof()){
		for(i=0;i<NUM_BH_SITES;i++){
			in3 >> myBHData->FDC2BH[n][i];
		}
		n++;
	}
	in3.close();
}
void vomitRoutes(route_sched *myRouteSched){
	int i,j;
	int totalMiles;

	cout << "Number of Routes: " << myRouteSched->numRoutes << endl;
	cout << "Best Iteration: " << myRouteSched->bestIter << endl;
	cout << "Lower Bound: " << myRouteSched->lowerBound << endl;

	ofstream out;
	out.open("ext/output.txt");
	if(!out){
		cout << "Cannot open file.\n";
	}
	for(i=0;i<myRouteSched->numRoutes;i++){
		out << "Route: " << i << endl;
		out << "Job*Start Time*Delivery Time*Return Time*Mileage"
<< endl;
		totalMiles=0;
		for(j=0;j<myRouteSched->bestRoutes[i].numRouteJobs; j++){
			totalMiles += myRouteSched-
>bestRoutes[i].JobList[j].rj_mile;
			out << myRouteSched->bestRoutes[i].JobList[j].rj_i <<
"*" << myRouteSched->bestRoutes[i].JobList[j].rj_start << "*" <<
myRouteSched->bestRoutes[i].JobList[j].rj_del << "*" << myRouteSched-
>bestRoutes[i].JobList[j].rj_ret << "*" << myRouteSched-
>bestRoutes[i].JobList[j].rj_mile << endl;
		}
		out << "***" << "Total Miles" << "*" << totalMiles << endl;
	}
	out.close();
}
void vomitRoutesAndBH(route_sched *myRouteSched){
	int i,j,k;
	int totalMiles;

	ofstream out;
	out.open("ext/alloutput.txt");
	if(!out){
		cout << "Cannot open file.\n";
	}

	for(i=0;i<myRouteSched->numRoutes;i++){
		out << "Route: " << i << endl;
		out << "Job*Start Time*Delivery Time*Return Time*Mileage"
<< endl;
		totalMiles=0;
```

```
            for(j=0;j<myRouteSched->bestRoutes[i].numRouteJobs; j++){
                  totalMiles += myRouteSched-
>bestRoutes[i].JobList[j].rj_mile;
                  out << myRouteSched->bestRoutes[i].JobList[j].rj_i <<
"*" << myRouteSched->bestRoutes[i].JobList[j].rj_start << "*" <<
myRouteSched->bestRoutes[i].JobList[j].rj_del << "*" << myRouteSched-
>bestRoutes[i].JobList[j].rj_ret << "*" << myRouteSched-
>bestRoutes[i].JobList[j].rj_mile << endl;
                  for(k=0;k<myRouteSched-
>bestRoutes[i].JobList[j].numLegBH;k++){
                        out << myRouteSched-
>bestRoutes[i].JobList[j].BHList[k].backhaulID << "*" << myRouteSched-
>bestRoutes[i].JobList[j].BHList[k].bhStart << "*" << myRouteSched-
>bestRoutes[i].JobList[j].BHList[k].loadSize << "**" << endl;
                        out << endl;
                  }
            }
            out << "***" << "Total Miles" << "*" << totalMiles << endl;
      }
      out.close();
}
void vomitRoutesAndInputSched(route_sched *myRouteSched, inp_sched
*myInpSched,char filename[]){
      int i,j;
      int jobIdx;
      ofstream out;

      out.open(filename);
      if(!out){
            cout << "Cannot open file.\n";
      }

      for(i=0;i<myRouteSched->numRoutes;i++){
            out << "Route: " << i << endl;

            for(j=0;j<myRouteSched->bestRoutes[i].numRouteJobs; j++){
                  jobIdx = myRouteSched->bestRoutes[i].JobList[j].rj_i;

                  out << "Job: " << jobIdx << "*" << "Leg: " << j <<
endl;
                  out << "Schedule*Early Start*Normal Start*Late
Start*Delivery Time*Return Time*Mileage" << endl;
                  out << "Input*" << myInpSched->inp_jobs[jobIdx].Ej <<
"*" << myInpSched->inp_jobs[jobIdx].Dj << "*" << myInpSched-
>inp_jobs[jobIdx].Lj << "*" << myInpSched->inp_jobs[jobIdx].Cj << "*"
<< myInpSched->inp_jobs[jobIdx].Rj << "*" << myInpSched-
>inp_jobs[jobIdx].Mj << endl;
                  //out << "Output*N/A*" << leftovers(myRouteSched-
>bestRoutes[i].JobList[j].rj_start,168) << "*N/A*" << myRouteSched-
>bestRoutes[i].JobList[j].rj_del << "*" << myRouteSched-
>bestRoutes[i].JobList[j].rj_ret << "*" << myRouteSched-
>bestRoutes[i].JobList[j].rj_mile << endl;
                  out << "Output*N/A*" << myRouteSched-
>bestRoutes[i].JobList[j].rj_start << "*N/A*" << myRouteSched-
>bestRoutes[i].JobList[j].rj_del << "*" << myRouteSched-
>bestRoutes[i].JobList[j].rj_ret << "*" << myRouteSched-
>bestRoutes[i].JobList[j].rj_mile << endl;
```

```
                  out << endl;

            }
      }
      out.close();
}
float getSolnRouteData(route_sched *myRouteSched,int data_type){
      int i;
      float giveBack;

      switch(data_type){
            case 1: //Number of routes with processing time < 50%
                  float numLowRoutes;
                  float rtePrHrs;

                  numLowRoutes = 0;
                  for(i=0;i<myRouteSched->numRoutes;i++){
                        rtePrHrs = getRouteProcessHours(&myRouteSched-
>bestRoutes[i]);
                        if((rtePrHrs/168)<0.50){
                              numLowRoutes++;
                        }
                  }
                  giveBack = numLowRoutes;
                  break;
      }
      return giveBack;
}
int rollDice(int rdmUpBnd){
      double r;
      int x;

      r = ((double)rand() / (double)(RAND_MAX+1));
      x = (int)(r*rdmUpBnd);

      return x;
}
int jobOnLB(job schedJobs[], int numSchedJobs, struct run_settings
*myRunSetup){
      int i,k,mod_k;
      float sngJobSt,sngJobFin;
      int hoursOfWeek[168]={0};
      int maxTrucks;
      int maxHour;

      for(i=0;i<numSchedJobs;i++){

            // If the job can be scheduled anytime (chng_ind==1) then
ignore it
            if(schedJobs[i].chng_ind != 1){
                  //Job is definitely on truck between latest start
time Lj and earliest complete time Ej + Cj + Rj
                  if(schedJobs[i].Lj < schedJobs[i].Ej){
                        sngJobSt = 168 + schedJobs[i].Lj;
                  }
                  else{
                        sngJobSt = schedJobs[i].Lj;
```

```
                  }
                  sngJobFin = schedJobs[i].Ej + schedJobs[i].Cj +
schedJobs[i].Rj + myRunSetup->LT + myRunSetup->TA;

                  if(sngJobSt < sngJobFin){
                        for(k=sngJobSt;k<sngJobFin;k++){
                              if(k<=167){
                                    hoursOfWeek[k]++;
                              }
                              else{
                                    mod_k = leftovers(k,168);
                                    hoursOfWeek[mod_k]++;
                              }
                        }
                  }
            }
      }

      maxTrucks = 0;
      maxHour = 0;
      for(k=0;k<168;k++){
            if(hoursOfWeek[k] > maxTrucks){
                  maxTrucks = hoursOfWeek[k];
            }
      }

      return maxTrucks;

}
int milesLB(job schedJobs[], int numSchedJobs, int maxMiles){
      int i;
      int totalMiles;
      float trucksNeeded;

      totalMiles = 0;
      for(i=0;i<numSchedJobs;i++){
            totalMiles += schedJobs[i].Mj;
      }

      trucksNeeded = totalMiles/maxMiles;

      return tceil(trucksNeeded);
}
int hoursLB(job schedJobs[], int numSchedJobs, struct run_settings
*myRunSettings){
      int i;
      float totalHours;
      float trucksNeeded;

      totalHours = 0;
      for(i=0;i<numSchedJobs;i++){
            totalHours += schedJobs[i].Cj + schedJobs[i].Rj +
myRunSettings->TA + myRunSettings->LT + myRunSettings->UT;
      }

      trucksNeeded = totalHours/myRunSettings->maxRtePHrs;
```

```
	return tceil(trucksNeeded);
}
float getLegBHLoad(route_job *gllRouteJob){
	float m_legLoad;
	int numBHs;
	int j;

	m_legLoad = 0;
	numBHs = gllRouteJob->numLegBH;

	if(numBHs>0){
		for(j=0;j<numBHs;j++){
			m_legLoad += gllRouteJob->BHList[j].loadSize;
		}
	}
	else{
		m_legLoad = 0;
	}

	return m_legLoad;
}
float getLegMileage(route_job *glmRouteJob){
	float m_legMiles;
	int numBHs;
	int j;

	m_legMiles = 0;
	numBHs = glmRouteJob->numLegBH;

	if(numBHs>0){
		m_legMiles = (int) glmRouteJob->rj_mile/2;
		m_legMiles += glmRouteJob->BHList[numBHs-1].milesToNDC;
		for(j=numBHs;j>0;j--){
			m_legMiles += glmRouteJob->BHList[j-1].milesToPickUp;
		}
	}
	else{
		m_legMiles = glmRouteJob->rj_mile;
	}

	return m_legMiles;
}
float getRouteMileageByLeg(route *grmRoute){
//    returns the total route mileage by leg (i.e. including backhauls)
rather than
//    by job
	float m_routeMiles;
	int i;

	m_routeMiles = 0;
	for(i=0; i<grmRoute->numRouteJobs; i++){
		m_routeMiles += getLegMileage(&grmRoute->JobList[i]);
	}

	return m_routeMiles;
}
float getRouteMileage(route *grmRoute){
```

```
	float m_routeMiles;
	int i;

	m_routeMiles = 0;
	for(i=0; i<grmRoute->numRouteJobs; i++){
		m_routeMiles = m_routeMiles + grmRoute->JobList[i].rj_mile;
	}

	return m_routeMiles;
}
float getLegPHours(route_job *glpRouteJob){
	float m_legHours;
	int numBHs;
	int j;

	m_legHours = 0;
	numBHs = glpRouteJob->numLegBH;

	if(numBHs>0){
		m_legHours = glpRouteJob->rj_legpt;

		for(j=numBHs;j>0;j--){
			m_legHours += glpRouteJob->BHList[j-1].milesToPickUp;
		}
	}
	else{
		m_legHours = glpRouteJob->rj_mile;
	}

	return m_legHours;
}
float getRouteProcessHours(route *grpRoute){
	float totalRoutePHours;
	int i;

	totalRoutePHours = 0;
	for(i=0; i<grpRoute->numRouteJobs; i++){
		totalRoutePHours = totalRoutePHours + grpRoute-
>JobList[i].rj_pt;
	}

	return totalRoutePHours;
}
int chooseNextJobByRandom(int arrUfeas[], int rNumUfeas){
	int myRoll;

	myRoll=rollDice(rNumUfeas-1);

	return arrUfeas[myRoll];
}
int chooseNextJobByPT(int ptMode, int arrUfeas[], float arrPi[], int
ptNumUfeas){
	float criteria[MAX_JOBS];
	int criteriaIdx[MAX_JOBS];
	int g;

	for(g=0; g<ptNumUfeas; g++){
```

```
            criteria[g]=arrPi[g];
            criteriaIdx[g]=arrUfeas[g];
      }

      sortArrayIdx(ptMode,criteria,criteriaIdx,ptNumUfeas);

      return criteriaIdx[0];
}
int chooseNextJobByStart(int stMode, int arrUfeas[], int stNumUfeas,
float arrESTimes[]){
      float criteria[MAX_JOBS];
      int criteriaIdx[MAX_JOBS];
      int g;

      for(g=0; g<stNumUfeas; g++){
            criteria[g]=arrESTimes[arrUfeas[g]];
            criteriaIdx[g]=arrUfeas[g];
      }

      sortArrayIdx(stMode,criteria,criteriaIdx,stNumUfeas);

      return criteriaIdx[0];
}
float calcTravelTime(int dist, int speed){
      float baseTime, remains, retTime;
      int wholeTime;

      baseTime = (float) dist/speed;
      wholeTime = (int) baseTime;

      remains = baseTime - wholeTime;
      if(remains > 0){
            if(remains < 0.5){
                  retTime = wholeTime + 0.5;
            }
            else{
                  retTime = wholeTime + 1.0;
            }
      }
      else{
            retTime = baseTime;
      }

      return retTime;
}
float calculateAdjustedStart(int rteIndex, int ji, float sngCki, job
*jobProps, float sngPi, float sngRteStart){
      /*
            rteIndex & ji were included for debugging help, they are no
longer needed.
      */

      float sngES, sngLS;     // TW earliest and latest stat time
      float sngEPS;                 // Max(Cki, sngES) = Earliest
POSSIBLE start time
      float sngExS, sngExF;   // Exclusion period start and finish
```

```
      // Adjust Earliest and Latest start time relative to the route
start time
      if(jobProps->chng_ind == 1){
            sngES = 0;
            sngLS = 336;
      }
      else{
            sngES = jobProps->Ej < sngRteStart ? jobProps->Ej + 168 :
jobProps->Ej;
            sngLS = jobProps->Lj < sngRteStart ? jobProps->Lj + 168 :
jobProps->Lj;
      }
      sngEPS = getMax(sngCki, sngES);

      // Adjust the Exclusion period relative to the route start time
      if(jobProps->ex_ind >= 1){
            sngExS = jobProps->ExAj < sngRteStart ? jobProps->ExAj +
168 : jobProps->ExAj;
            sngExF = jobProps->ExBj < sngRteStart ? jobProps->ExBj +
168 : jobProps->ExBj;
      }

      // Step 1: Check the job's latest departure time
      if(sngLS < sngCki){
            return -1;
      }

      // Step 2: Does the job Pi allow it to fit in the amount of time
left on the schedule if it left as early as possible
      if(sngEPS + sngPi > sngRteStart + 168){
            return -2;
      }

      // Step 3: If there is an exclusion period, when can I start the
job?
      if(jobProps->ex_ind >= 1){
            if(sngEPS + jobProps->Cj || sngEPS + jobProps->Cj >
sngExF){
                  return sngEPS;
            }
            else if(sngExF - jobProps->Cj < sngLS){
                  return sngExF - jobProps->Cj;
            }
            else{
                  return -3;
            }
      }

      // Step 4: There isn't an exclusion period, and we passed all of
the regular checks so just return sngEPS
      return sngEPS;
}
float calculateAdjustedPullStart(float sngCki, job *jobProps, float
sngPi, float sngRteStart){
/*
      Note: This function ignores Exclusion periods for simplification
```

```
      sngCki: completion time of the job preceeding the current job
(gapStart)

      Based on job properties Ej,Lj (earliest/latest start times), this
function will calculate the earliest
      start time for the job based on the jobs that come before it on
the route.
*/
      float sngES;                  // TW earliest start time
      float sngEPS;                 // Earliest possible start time for
job

      if(jobProps->chng_ind == 1){//Job can be started anytime
            //We'll just start it at the completion of the preceding
job
            sngEPS = sngCki;
      }
      else{//Job has a time window (TW can be 0)
            //Adjust the job's ES/LS for this specific route.
            sngES = jobProps->Ej < sngRteStart ? jobProps->Ej + 168 :
jobProps->Ej;
           //sngLS = (jobProps->Lj < sngRteStart ? jobProps->Lj + 168 :
jobProps->Lj);

            //The job's earliest possible start time is the max of
sngCki and then beginning of the job's departure time window
            sngEPS = getMax(sngCki,sngES);
      }

      return sngEPS;
}
float calculateAdjustedPushStart(float sngSkj, job *jobProps, float
sngPi, float sngRteStart){
/*
      Note: This function ignores Exclusion periods for simplification

      *OMITTED* sngCki: completion time of the job preceeding the
current job (gapStart)
      sngSkj: start time of the job following the current job (gapEnd)

      Based on job properties Ej,Lj (earliest/latest start times), this
function will calculate the latest
      possible start time for the job based on the gap it is currently
trying to fit in.
*/
      float sngLS;                  // TW earliest and latest stat time
      float sngLPS;                 // Latest possible start time for
job
      float gapLate;                // Latest time the job can start
and avoid overlapping successive job

      //Both scenarios use gapLate, so calculate now.
      gapLate = sngSkj - sngPi;

      // Caclulate LPS based on what type of time window the job has
      if(jobProps->chng_ind == 1){//Job can be scheduled anytime
```

```
            //Since the job can be scheduled anytime, we don't need to
make any adjustments to the job's ES/LS
            sngLPS = gapLate;

      }
      else{//Job has a time window (tw can be 0)

            //Adjust the job's ES/LS for this specific route.
            //sngES = jobProps->Ej < sngRteStart ? jobProps->Ej + 168 :
jobProps->Ej;
            sngLS = (jobProps->Lj < sngRteStart ? jobProps->Lj + 168 :
jobProps->Lj);

            //The job's latest possible start time is the min of
gapLate and the end of the job's departure time window
            sngLPS = getMin(gapLate,sngLS);
      }

      return sngLPS;
}
void addJobToRoute(struct job *nJob,float nJobPi, int nJobIdx, float
nJobStart, struct route *cRoute, int routeLeg, struct run_settings
*myRunSetup){

      cRoute->JobList[routeLeg].rj_i=nJobIdx;
      cRoute->JobList[routeLeg].rj_start=nJobStart;
      cRoute->JobList[routeLeg].rj_del=cRoute-
>JobList[routeLeg].rj_start + nJob->Cj;
      cRoute->JobList[routeLeg].rj_ret=cRoute->JobList[routeLeg].rj_del
+ myRunSetup->LT + nJob->Rj;
      cRoute->JobList[routeLeg].rj_mile=nJob->Mj;
      cRoute->JobList[routeLeg].resched_ind = 0;
      cRoute->JobList[routeLeg].rj_pt=nJobPi;
      cRoute->JobList[routeLeg].numLegBH = 0;

      cRoute->JobList[routeLeg].job_props = *nJob;
}
int packBackHauls(struct route_sched *packRoutes, struct bh_sched
*myBHSched, struct run_settings *myRunSetup, struct bh_data *myBHData){
//    Attempts to add backhauls to the route schedule.  Returns the
number of
//    backhauls successfully added to the route schedule.
      int i,j,k;
      int numBackHauls, numRoutes, numRouteLegs;
      int numLegBH;
      int countBH;
      int dist, retDist;
      int nBHLoc, pLoc;
      int bhSchedIndicator;

      float nBHLoad,nBHStart,pRetMiles, pRetTime;
      float travelToTime,retTime,nBHPt;
      float nextLegStart, routeStart, routeEnd;

      float checkMiles, checkHours, checkLoad;

      numBackHauls = myBHSched->numBH;
```

```
      numRoutes = packRoutes->numRoutes;

//    Now try assigning the bh jobs to the routes.
      countBH = 0;
      for(i=0;i<numBackHauls;i++){
            bhSchedIndicator=0;

            nBHLoc = myBHSched->inp_bh[i].locID;
            nBHLoad = myBHSched->inp_bh[i].loadSize;

            for(j=0;j<numRoutes;j++){

                  if (bhSchedIndicator) break;

                  numRouteLegs=packRoutes->bestRoutes[j].numRouteJobs;
                  routeStart = packRoutes-
>bestRoutes[j].JobList[0].rj_start;
                  routeEnd = routeStart + 168;

                  for(k=0;k<numRouteLegs;k++){
                        numLegBH=packRoutes-
>bestRoutes[j].JobList[k].numLegBH;

                        if(numLegBH>MAX_LEG_BH) continue;

                        if(numLegBH>0){
//
      ******************************************
//                          *             PERFORMANCE MODIFICATION
          *
//                          * Test BH's are full load, so numLegBH !>
1 *
//                          * so I skip this section, and just
continue *
//
      ******************************************
                          continue;
                          /*
                          pLoc = packRoutes-
>bestRoutes[j].JobList[k].BHList[numLegBH-1].locationID;
                          pRetMiles = packRoutes-
>bestRoutes[j].JobList[k].BHList[numLegBH-1].milesToNDC;
                          pRetTime = packRoutes-
>bestRoutes[j].JobList[k].BHList[numLegBH-1].bhRt;

                          //The start time for the new backhaul is
the start time of the previous bh + travel time to that backhaul +
loading time
                          nBHStart = packRoutes-
>bestRoutes[j].JobList[k].BHList[numLegBH-1].bhStart + packRoutes-
>bestRoutes[j].JobList[k].BHList[numLegBH-1].bhCt + myRunSetup->LT;

                          checkLoad = getLegBHLoad(&packRoutes-
>bestRoutes[j].JobList[k]) + nBHLoad;

                          //distance is the BH2BH[cBHLoc,nBHLoc]
distance
```

```
                              dist = myBHData->BH2BH[pLoc][nBHLoc];

                              if(dist<=0) continue; //Means you can't
travel between these two backhauls
                              */
                        }
                        else{
                              pLoc = packRoutes-
>bestRoutes[j].JobList[k].job_props.fdcIdx;
                              pRetMiles = packRoutes-
>bestRoutes[j].JobList[k].job_props.Mj / 2;
                              pRetTime = packRoutes-
>bestRoutes[j].JobList[k].rj_ret;

                              //The start time for the new backhaul is
the delivery arrival time + system loading/unloading attribute
                              nBHStart = packRoutes-
>bestRoutes[j].JobList[k].rj_del + myRunSetup->LT;

                              checkLoad = nBHLoad;

                              //distance is the FDC2BH[FDC,nBHLoc]
distance
                              dist = myBHData->FDC2BH[pLoc][nBHLoc];

                              if(dist<=0) continue; //Means BH is
"practically" unreachable from this FDC
                        }
                        //Return distance is the BH2NDC[nBHLoc]
distance
                        retDist = myBHData->BH2NDC[nBHLoc];

                        travelToTime=calcTravelTime(dist,myRunSetup-
>mph);
                        retTime=calcTravelTime(dist,myRunSetup->mph);

                        checkMiles = getRouteMileage(&packRoutes-
>bestRoutes[j]) - pRetMiles + dist + retDist;
                        checkHours = getRouteProcessHours(&packRoutes-
>bestRoutes[j]) - pRetTime + travelToTime + retTime + myRunSetup->LT;

                        if(checkMiles <= myRunSetup->maxMiles &&
checkHours <= myRunSetup->maxRtePHrs && checkLoad <= myRunSetup-
>maxLoadSize){
                              nextLegStart = (k != numRouteLegs-1 ?
packRoutes->bestRoutes[j].JobList[k+1].rj_start : routeEnd);

                              nBHPt = travelToTime + retTime +
myRunSetup->LT + myRunSetup->TA;
                              if(nBHStart + nBHPt < nextLegStart){
                                    //Well it passes all the tests, so
I should be able to add it.
                                    numLegBH++;
                                    packRoutes-
>bestRoutes[j].JobList[k].numLegBH=numLegBH;
                                    packRoutes-
>bestRoutes[j].JobList[k].BHList[numLegBH-1].backhaulID=i;
```

```
                                    packRoutes-
>bestRoutes[j].JobList[k].BHList[numLegBH-1].locationID=nBHLoc;
                                    packRoutes-
>bestRoutes[j].JobList[k].BHList[numLegBH-1].milesToPickUp=dist;
                                    packRoutes-
>bestRoutes[j].JobList[k].BHList[numLegBH-1].milesToNDC=retDist;
                                    packRoutes-
>bestRoutes[j].JobList[k].BHList[numLegBH-1].bhCt=travelToTime;
                                    packRoutes-
>bestRoutes[j].JobList[k].BHList[numLegBH-1].bhRt=retTime;
                                    packRoutes-
>bestRoutes[j].JobList[k].BHList[numLegBH-1].loadSize=nBHLoad;
                                    packRoutes-
>bestRoutes[j].JobList[k].BHList[numLegBH-1].bhStart=nBHStart;

                                    //Now adjust the necessary route
leg data (miles & hours)
                                    packRoutes-
>bestRoutes[j].JobList[k].rj_mile = getLegMileage(&packRoutes-
>bestRoutes[j].JobList[k]);
                                    packRoutes-
>bestRoutes[j].JobList[k].rj_pt = getLegPHours(&packRoutes-
>bestRoutes[j].JobList[k]);

                                    //I'm not sold on this, but
temporarily at least, I am updating rj_ret;
                                    packRoutes-
>bestRoutes[j].JobList[k].rj_ret = nBHStart + nBHPt;

                                    //Now figure out how to skip to the
next backhaul
                                    bhSchedIndicator=1;
                                    countBH++;
                                    break;
                                }
                            }
                        }//numRouteLegs
                    }//numRoutes
            }//numBackHauls

            return countBH;
}
void fixedBHLite(struct inp_sched *myInpSched, struct run_settings
*myRunSetup, struct bh_data *myBHData, int pnum,char filename[]){
            int numJobs, numUfeas;
            int Bestsol, Ufeas[MAX_JOBS];
            float bestStart[MAX_JOBS], Pi[MAX_JOBS];
            int unrouted[MAX_JOBS], numUnrouted, rteIdx;
            int diceRoll;
            float cki;
            int k0, ki, kii, kj, remIdx;
            int i, j, k, m, q, g, t;
            int result[4][1000];
            int bhResult[6][1000];
            long s;
            ofstream out;
            ofstream bhout;
```

```
	float routeStartTime;

	struct route_sched bestRouteSoln;
	struct route_sched testRouteSoln;
	struct route currRoutes[MAX_ROUTES];
	struct bh_sched checkBHSystem[6];
	int bhChecker;
	int bhSizes[6] = {5,10,20,40,80,100};
	int numCurrentRouteJobs;

	float checkMiles;
	float checkRtePHours;

	char bhInputFile[80];
	char bhOutputFile[80];

	numJobs = myInpSched->numSchedJobs;
	Bestsol = numJobs;

	//Grab backhaul problems
	for(k=0;k<6;k++){

	sprintf(bhInputFile,"../probs/%d/bh/%d_%d_%04d.txt",numJobs,numJo
bs,bhSizes[k],pnum);
		grabBHProblem(&checkBHSystem[k],bhInputFile);
	}
	for(t=0; t<4; t++){
		//Run numIters randomized iterations of the algorithm
		for(s=0; s<myRunSetup->numIters; s++){
			numUnrouted = numJobs;
			numUfeas = 0;
			numCurrentRouteJobs = 1;

			//Initialize Pi() data
			for(i=0; i<numJobs; i++){
				Pi[i] = myInpSched->inp_jobs[i].Cj +
myInpSched->inp_jobs[i].Rj + myRunSetup->TA + myRunSetup->LT +
myRunSetup->UT;
				unrouted[i]=i;
			}

			//Step 1: Randomly assign the first job on the first
route.
			rteIdx = 0;
			diceRoll = rollDice(numUnrouted-1);
			k0=unrouted[diceRoll];

			currRoutes[rteIdx].routeStatus = 1;
		       currRoutes[rteIdx].numRouteJobs  =  numCurrentRouteJobs;

			addJobToRoute(&myInpSched-
>inp_jobs[k0],Pi[k0],k0,myInpSched-
>inp_jobs[k0].Dj,&currRoutes[rteIdx],0,myRunSetup);

			// Get the unrouted[] index of the assigned job and
remove it from the list of unrouted jobs
```

```
                  remIdx = arrFindValue(unrouted,numUnrouted,k0);
                  numUnrouted = arrRemoveByIndex(unrouted, numUnrouted,
remIdx);

                  numUfeas = 0;
                  //numUnrouted already updated

                  while(numUnrouted>0){
                        numUfeas=0;       // Set the number of feasible
jobs to zero.
                        j = numCurrentRouteJobs-1;
                        routeStartTime =
currRoutes[rteIdx].JobList[0].rj_start;   // start time of first job on
route
                        ki = currRoutes[rteIdx].JobList[j].rj_i;
                  // job index of last job on route
                        cki = currRoutes[rteIdx].JobList[j].rj_start +
Pi[ki];           // Completion time of last job on route

                        // Feasibility check of unrouted jobs on
current route.
                        for(m=0; m<numUnrouted; m++){
                              bestStart[unrouted[m]] =
calculateAdjustedStart(rteIdx, unrouted[m], cki, &myInpSched-
>inp_jobs[unrouted[m]], Pi[unrouted[m]], routeStartTime);
                              checkMiles =
getRouteMileage(&currRoutes[rteIdx]);
                              checkRtePHours =
getRouteProcessHours(&currRoutes[rteIdx]);

                              if(bestStart[unrouted[m]] >= 0 &&
checkMiles + myInpSched->inp_jobs[unrouted[m]].Mj <= myRunSetup-
>maxMiles && checkRtePHours + Pi[unrouted[m]] <= myRunSetup->maxRtePHrs
&& numCurrentRouteJobs < MAX_ROUTE_JOBS){
                                    //Job is feasible, do stuff
                                    Ufeas[numUfeas] = unrouted[m];
                                    numUfeas++;
                              }
                        }
                        if(numUfeas>0){
                              switch(t){
                                    case 0:
                                          // Choose next job by longest
processing time
                                          kii =
chooseNextJobByPT(0,Ufeas,Pi,numUfeas);
                                          break;
                                    case 1:
                                          // Choose next job by
shortest processing time
                                          kii =
chooseNextJobByPT(1,Ufeas,Pi,numUfeas);
                                          break;
                                    case 2:
                                          // Choose next job by
earliest start time
```

```
                                        kii =
chooseNextJobByStart(1,Ufeas,numUfeas,bestStart);
                                        break;
                                  case 3:
                                        // Choose next job randomly
                                        kii =
chooseNextJobByRandom(Ufeas,numUfeas);
                                        break;
                                  default:
                                        // Default: choose job by
longest processing time
                                        kii =
chooseNextJobByPT(0,Ufeas,Pi,numUfeas);
                                        break;
                              }
                              numCurrentRouteJobs++;

                              //Add the chosen job to the current route
                              currRoutes[rteIdx].numRouteJobs =
numCurrentRouteJobs;
                              j = numCurrentRouteJobs-1;

                              addJobToRoute(&myInpSched-
>inp_jobs[kii],Pi[kii],kii,bestStart[kii],&currRoutes[rteIdx],j,myRunSe
tup);

                              // Get the unrouted[] index of the
assigned job and remove it from the list of unrouted jobs
                              remIdx =
arrFindValue(unrouted,numUnrouted,kii);
                              numUnrouted = arrRemoveByIndex(unrouted,
numUnrouted, remIdx);
                        }
                        else{//(numUfeas<=0)
                              // Start a new route and randomly choose
a job to start the route
                              rteIdx++;

                              /* If the number of routes is greater or
equal to our best solution then
                              just skip the rest of this iteration to
save computation time*/
                              //if(rteIdx>=Bestsol) goto SkipIteration;

                              numCurrentRouteJobs=1;

      currRoutes[rteIdx].numRouteJobs=numCurrentRouteJobs;

                              kj=rollDice(numUnrouted-1);
                              kii=unrouted[kj];

                              j=0;
                              currRoutes[rteIdx].routeStatus = 1;
      //The route is active
                              currRoutes[rteIdx].numRouteJobs=1;
```

```
                              addJobToRoute(&myInpSched-
>inp_jobs[kii],Pi[kii],kii,myInpSched-
>inp_jobs[kii].Dj,&currRoutes[rteIdx],j,myRunSetup);

                              // Get the unrouted[] index of the
assigned job and remove it from the list of unrouted jobs
                              numUnrouted = arrRemoveByIndex(unrouted,
numUnrouted, kj);

                        }
                  }//while(numUnrouted>0)

                  result[t][s] = rteIdx+1;

                  for(g=0; g<rteIdx; g++){
                        bestRouteSoln.bestRoutes[g]=currRoutes[g];
                  }
                  bestRouteSoln.numRoutes = rteIdx+1;

                  //Process backhauls, and see how many (total load)
are fulfilled
                  bhChecker=0;
                  for(k=0;k<6;k++){
                        testRouteSoln=bestRouteSoln;


      bhChecker=packBackHauls(&testRouteSoln,&checkBHSystem[k],myRunSet
up,myBHData);
                        bhResult[k][s]=bhChecker;
                  }
            }//(s=0; s<myRunSetup->numIters; s++)

            //Output bh data for method t here

      sprintf(bhOutputFile,"../solutions/%d/bh/%d_m%d_%04d.txt",numJobs
,numJobs,t+1,pnum);
            bhout.open(bhOutputFile);
            for(q=0;q<myRunSetup->numIters;q++){
                  for(k=0;k<6;k++){
                        bhout << bhResult[k][q];
                        if(k!=5){
                              bhout << "\t";
                        }
                  }
                  bhout << endl;
            }
            bhout.close();
      }//for(t=0;t<4;t++);

      out.open(filename);
      if(!out){
            cout << "Cannot open solution output file\n" << endl;
      }
      for(s=0;s<myRunSetup->numIters;s++){
            for(t=0;t<4;t++){
                  //output results to a file.
                  out << result[t][s];
```

```
                  if(t!=3){
                        out << "\t";
                  }
            }
            out << endl;
      }
      out.close();
}
void fixedNoBH(struct inp_sched *myInpSched, struct route_sched
*myRouteSched, struct run_settings *myRunSetup){
      int numJobs, numUfeas;
      int Bestsol, Ufeas[MAX_JOBS];
      float bestStart[MAX_JOBS], Pi[MAX_JOBS];
      int unrouted[MAX_JOBS], numUnrouted, rteIdx;
      int diceRoll;
      float cki;
      int k0, ki, kii, kj, remIdx;
      int i, j, m, g;
      long s;

      float routeStartTime;

      struct route currRoutes[MAX_ROUTES];
      int numCurrentRouteJobs;

      float checkMiles;
      float checkRtePHours;

      numJobs = myInpSched->numSchedJobs;
      Bestsol = numJobs;

      myRouteSched->lowerBound = jobOnLB(myInpSched-
>inp_jobs,myInpSched->numSchedJobs,myRunSetup);

      //Run numIters randomized iterations of the algorithm
      for(s=0; s<myRunSetup->numIters; s++){
            numUnrouted = numJobs;
            numUfeas = 0;
            numCurrentRouteJobs = 1;
            //Initialize Pi() data
            for(i=0; i<numJobs; i++){
                  Pi[i] = myInpSched->inp_jobs[i].Cj + myInpSched-
>inp_jobs[i].Rj + myRunSetup->TA + myRunSetup->LT + myRunSetup->UT;
                  unrouted[i]=i;
            }

            //Step 1: Randomly assign the first job on the first route.
            rteIdx = 0;
            diceRoll = rollDice(numUnrouted-1);
            k0=unrouted[diceRoll];
            currRoutes[rteIdx].routeStatus = 1;
            currRoutes[rteIdx].numRouteJobs = numCurrentRouteJobs;

            addJobToRoute(&myInpSched-
>inp_jobs[k0],Pi[k0],k0,myInpSched-
>inp_jobs[k0].Dj,&currRoutes[rteIdx],0,myRunSetup);
```

```
            // Get the unrouted[] index of the assigned job and remove
it from the list of unrouted jobs
            remIdx = arrFindValue(unrouted,numUnrouted,k0);
            numUnrouted = arrRemoveByIndex(unrouted, numUnrouted,
remIdx);
            numUfeas = 0;
            //numUnrouted already updated

            while(numUnrouted>0){
                  numUfeas=0;       // Set the number of feasible jobs
to zero.
                  j = numCurrentRouteJobs-1;
                  routeStartTime =
currRoutes[rteIdx].JobList[0].rj_start;   // start time of first job on
route
                  ki = currRoutes[rteIdx].JobList[j].rj_i;
            // job index of last job on route
                    cki = currRoutes[rteIdx].JobList[j].rj_start + Pi[ki];
            // Completion time of last job on route

                  // Feasibility check of unrouted jobs on current
route.
                  for(m=0; m<numUnrouted; m++){
                        bestStart[unrouted[m]] =
calculateAdjustedStart(rteIdx, unrouted[m], cki, &myInpSched-
>inp_jobs[unrouted[m]], Pi[unrouted[m]], routeStartTime);
                        checkMiles =
getRouteMileage(&currRoutes[rteIdx]);
                        checkRtePHours =
getRouteProcessHours(&currRoutes[rteIdx]);

                        if(bestStart[unrouted[m]] >= 0 && checkMiles +
myInpSched->inp_jobs[unrouted[m]].Mj <= myRunSetup->maxMiles &&
checkRtePHours + Pi[unrouted[m]] <= myRunSetup->maxRtePHrs &&
numCurrentRouteJobs < MAX_ROUTE_JOBS){
                              //Job is feasible, do stuff
                              Ufeas[numUfeas] = unrouted[m];
                              numUfeas++;
                        }
                  }
                  if(numUfeas>0){
                        switch(myRunSetup->selCrit){
                              case 0:
                                    // Choose next job by longest
processing time
                                    kii =
chooseNextJobByPT(0,Ufeas,Pi,numUfeas);
                                    break;
                              case 1:
                                    // Choose next job by shortest
processing time
                                    kii =
chooseNextJobByPT(1,Ufeas,Pi,numUfeas);
                                    break;
                              case 2:
                                    // Choose next job by earliest
start time
```

```
                                    kii =
chooseNextJobByStart(1,Ufeas,numUfeas,bestStart);
                                    break;
                              case 3:
                                    // Choose next job randomly
                                    kii =
chooseNextJobByRandom(Ufeas,numUfeas);
                                    break;
                              default:
                                    // Default: choose job by longest
processing time
                                    kii =
chooseNextJobByPT(0,Ufeas,Pi,numUfeas);
                                    break;
                        }
                        numCurrentRouteJobs++;

                        //Add the chosen job to the current route
                        currRoutes[rteIdx].numRouteJobs =
numCurrentRouteJobs;
                        j = numCurrentRouteJobs-1;

                        addJobToRoute(&myInpSched-
>inp_jobs[kii],Pi[kii],kii,bestStart[kii],&currRoutes[rteIdx],j,myRunSe
tup);

                        // Get the unrouted[] index of the assigned job
and remove it from the list of unrouted jobs
                       remIdx  =  arrFindValue(unrouted,numUnrouted,kii);
                        numUnrouted = arrRemoveByIndex(unrouted,
numUnrouted, remIdx);
                  }
                  else{//(numUfeas<=0)
                        // Start a new route and randomly choose a job
to start the route
                        rteIdx++;

                        /* If the number of routes is greater or equal
to our best solution then
                        just skip the rest of this iteration to save
computation time*/
                        if(rteIdx>=Bestsol) goto SkipIteration;

                        numCurrentRouteJobs=1;

      currRoutes[rteIdx].numRouteJobs=numCurrentRouteJobs;
                        kj=rollDice(numUnrouted-1);
                        kii=unrouted[kj];
                        j=0;
                        currRoutes[rteIdx].routeStatus = 1;       //The
route is active
                        currRoutes[rteIdx].numRouteJobs=1;

                        addJobToRoute(&myInpSched-
>inp_jobs[kii],Pi[kii],kii,myInpSched-
>inp_jobs[kii].Dj,&currRoutes[rteIdx],j,myRunSetup);
```

```
                        // Get the unrouted[] index of the assigned job
and remove it from the list of unrouted jobs
                        numUnrouted = arrRemoveByIndex(unrouted,
numUnrouted, kj);
                  }
            }//while(numUnrouted>0)

            if(rteIdx+1<Bestsol){
                  Bestsol = rteIdx+1;
                  //Assign best route solution to the output struct
                  for(g=0; g<=rteIdx; g++){
                        myRouteSched->bestRoutes[g]=currRoutes[g];
                  }
                  myRouteSched->numRoutes=rteIdx+1;
                  myRouteSched->bestIter=s;
            }
            SkipIteration:;
      }
}
void insertJobs(struct inp_sched *myInpSched, struct route_sched
*myRouteSched, struct run_settings *myRunSetup){
      int i;
      int uninserted[MAX_JOBS], numUninsert;
      int insertSuccess;

      //Find the jobs in myInpSched that need to be
rescheduled/inserted
      numUninsert=0;
      for(i=0;i<myInpSched->numSchedJobs;i++){
            if(myInpSched->inp_jobs[i].resched_ind==1){
                  uninserted[numUninsert]=i;
                  numUninsert++;
            }
      }
      //Loop through uninserted[] jobs, and try to find a place for
them
      for(i=0;i<numUninsert;i++){

      insertSuccess=threesCompany(uninserted[i],myInpSched,myRouteSched
,myRunSetup);
            //if insertSuccess fails (i.e. = 0), should I add another
Route to the schedule or throw the job away?
      }
}
int threesCompany(int insJobIdx,struct inp_sched *myInpSched, struct
route_sched *myRouteSched,struct run_settings *myRunSetup){
      int i;
      int jobFit;
      float checkRteMiles, checkRteHours, insJobPi;

      insJobPi = myInpSched->inp_jobs[insJobIdx].Cj + myInpSched-
>inp_jobs[insJobIdx].Rj + myRunSetup->TA + myRunSetup->LT + myRunSetup-
>UT;
      for(i=0;i<myRouteSched->numRoutes;i++){
            checkRteMiles=getRouteMileage(&myRouteSched->bestRoutes[i])
+ myInpSched->inp_jobs[insJobIdx].Mj;
```

```
            checkRteHours=getRouteProcessHours(&myRouteSched-
>bestRoutes[i]) + insJobPi;
            if(checkRteMiles <= myRunSetup->maxMiles && checkRteHours
<= myRunSetup->maxRtePHrs && myRouteSched->bestRoutes[i].numRouteJobs <
MAX_ROUTE_JOBS){
                  //Route passes the "quick" checks to see if insertJob
fits, now the real test comes
                  //use simpleGapCheck();
                  jobFit = pushPullGapCheck(&myRouteSched-
>bestRoutes[i],i,&myInpSched->inp_jobs[insJobIdx],insJobPi);
                  if(jobFit >= 0){
                        return jobFit;
                  }
            }
      }
      //No route could fit this job on the schedule, so give up
      cout << "I was unable to fit job " << insJobIdx << " on an
existing route" << endl;
      return -1;
}
void pushRoute(route *clayRoute, int frzJobIdx, float rteEndTime, float
rteStartTime, int debugrteIdx){
      int j;
      int lastRteJobIdx;
      float pushEnd;
      float pushedStartTime, pst_diff;

      lastRteJobIdx = clayRoute->numRouteJobs-1;
      for(j=lastRteJobIdx;j>frzJobIdx;j--){
            pushEnd = (j != lastRteJobIdx ? clayRoute-
>JobList[j+1].rj_start : rteEndTime);
            pushedStartTime = calculateAdjustedPushStart(pushEnd,
&clayRoute->JobList[j].job_props,clayRoute-
>JobList[j].rj_pt,rteStartTime);

//          I didn't want to have to carry the complete job data and
run settings all the way through to here, so I have "jerry-rigged" the
//          reassignment of the job's on-route data.
           pst_diff = pushedStartTime - clayRoute->JobList[j].rj_start;
            if(pst_diff > 0){
                  clayRoute->JobList[j].rj_start += pst_diff;
                  clayRoute->JobList[j].rj_del += pst_diff;
                  clayRoute->JobList[j].rj_ret += pst_diff;
            }
      }
}
void pullRoute(route *clayRoute, int frzJobIdx, float rteEndTime, float
rteStartTime, int debugrteIdx){
      //frzJobIdx is the JobList idx of the *newly inserted job*
      int j;
      int lastRteJobIdx;
      float pullBegin;
      float pulledStartTime, pst_diff;

      lastRteJobIdx = clayRoute->numRouteJobs-1;
      for(j=frzJobIdx+1;j<=lastRteJobIdx;j++){
```

```
            pullBegin = clayRoute->JobList[j-1].rj_start + clayRoute-
>JobList[j-1].rj_pt;
            pulledStartTime = calculateAdjustedPullStart(pullBegin,
&clayRoute->JobList[j].job_props,clayRoute->JobList[j].rj_pt,
rteStartTime);

           pst_diff = pulledStartTime - clayRoute->JobList[j].rj_start;
            if(pst_diff < 0){
                  clayRoute->JobList[j].rj_start += pst_diff;
                  clayRoute->JobList[j].rj_del += pst_diff;
                  clayRoute->JobList[j].rj_ret += pst_diff;
            }
      }
}
int pushPullGapCheck(route *checkRoute, int rteIdx, job *newJob, float
newJobPi){
//    Called by threesCompany, returns the index of the route job the
new job will fit after, -1 if the job will not fit
//    on any routes.
      float routeStart, routeMaxTime;
      int i, statusInsert;
      route rtePlaydoh;
      float gapSize, gapStart, gapEnd, gapLate;
      float adjJobEarly, adjJobLate;

//    Copy the route so we can squish and squeeze it as we please.
      rtePlaydoh = *checkRoute;
      routeStart = rtePlaydoh.JobList[0].rj_start;
      routeMaxTime = routeStart + 168;

//   Iterate through all of the jobs on the route, except the last one,
setting job i as the "frozen" starting job
      for(i=0;i<rtePlaydoh.numRouteJobs;i++){
            if(checkRoute->routeStatus == 1){//Make sure the route
isn't "broken"
//                Step 1: "Push" all jobs after job i as late as they
will go.
                  pushRoute(&rtePlaydoh, i, routeMaxTime, routeStart,
rteIdx);

//                Step 2: Establish gap properties
                  //The gapStart time is the completion time of the
previous job plus the Turnaround time
                  //This is NOT the job's return time, so we must
calculate using the job's start time + the processing time.
                  gapStart = rtePlaydoh.JobList[i].rj_start +
rtePlaydoh.JobList[i].rj_pt;

                  //gapEnd is different if i is the index for the last
job on the route
                  gapEnd = (i != rtePlaydoh.numRouteJobs-1 ?
rtePlaydoh.JobList[i+1].rj_start : routeMaxTime);

                  gapSize = gapEnd - gapStart;        //The size of the
gap is the difference between gapStart and gapEnd
                  gapLate = gapEnd - newJobPi;        //The latest the
job can start and still fit in the gap
```

```
                  adjJobEarly =
calculateAdjustedStart(0,i,gapStart,newJob,newJobPi,routeStart);
                  adjJobLate = (newJob->Lj < routeStart ? newJob->Lj +
168 : newJob->Lj);

//                Step 3: Check gap availability.  If gap available,
insert job into temporary route
                  /*
                        Conditions of fit: (as they appear in if
statement)
                              1. Does calculateAdjustedStart return a
valid start time (one that occurs after Cki and falls in the job's time
window)
                              2. Is the size of the gap > the job's
processing time
                              3. Does the start time for the job at the
end of the gap occur after the "inserted" job is completed?
                  */
                  if(adjJobEarly >= 0 && gapSize >= newJobPi &&
adjJobEarly + newJobPi <= gapEnd){
                        //Insert job into the gap.
                        statusInsert =
simpleGapInsert(&rtePlaydoh,i,newJob,adjJobEarly,newJobPi,2);
                  }

//                Step 4: If job was inserted, then , "pull" route, and
copy back to checkRoute
                  if(statusInsert >= 0){
                        cout << "I just placed job " << newJob->idx <<
" on route " << rteIdx << " after leg " << i << " using
pushPullGapCheck." << endl;


      pullRoute(&rtePlaydoh,statusInsert+1,routeMaxTime,routeStart,rteI
dx);
                        *checkRoute = rtePlaydoh;

                        return statusInsert;
                  }
            }//if(checkRoute->routeStatus == 1)
      }//for(i=0;i<rtePlaydoh.numRouteJobs;i++)

      return -1;
}
int simpleGapCheck(route *checkRoute,int rteIdx, job *newJob, float
newJobPi){
// Called by threesCompany, returns the index of the route job the new
job will fit after, -1 if the job will not fit
// on any routes.
      float routeStart, routeMaxTime;
      int i, statusInsert;
      float gapSize, gapStart, gapEnd, gapLate;
      float adjJobEarly, adjJobLate;

      routeStart = checkRoute->JobList[0].rj_start;
      routeMaxTime = routeStart + 168;
```

```
      for(i=0;i<checkRoute->numRouteJobs;i++){
            if(checkRoute->routeStatus == 1){//Make sure the route
isn't "broken"
                  //The gapStart time is the completion time of the
previous job plus the Turnaround time
                  //This is NOT the job's return time, so we must
calculate using the job's start time + the processing time.
                  gapStart = checkRoute->JobList[i].rj_start +
checkRoute->JobList[i].rj_pt;

                  //gapEnd is different if i is the index for the last
job on the route
                  gapEnd = (i != checkRoute->numRouteJobs-1 ?
checkRoute->JobList[i+1].rj_start : routeMaxTime);

                  gapSize = gapEnd - gapStart;        //The size of the
gap is the difference between gapStart and gapEnd
                  gapLate = gapEnd - newJobPi;        //The latest the
job can start and still fit in the gap

                  adjJobEarly =
calculateAdjustedStart(0,i,gapStart,newJob,newJobPi,routeStart);
                  adjJobLate = (newJob->Lj < routeStart ? newJob->Lj +
168 : newJob->Lj);

                  if(adjJobEarly >= 0){
                        cout << "Route: " << rteIdx << endl;
                        cout << "Route Leg: " << i << endl;
                        cout << "Gap Start: " << gapStart << endl;
                        cout << "Gap End: " << gapEnd << endl;
                  }

                  /*
                        Conditions of fit: (as they appear in if
statement)
                              1. Does calculateAdjustedStart return a
valid start time (one that occurs after Cki and falls in the job's time
window)
                              2. Is the size of the gap > the job's
processing time
                              3. Does the start time for the job at the
end of the gap occur after the "inserted" job is completed?
                  */
                  if(adjJobEarly >= 0 && gapSize >= newJobPi &&
adjJobEarly + newJobPi <= gapEnd){
                        //Insert job into the gap.
                        statusInsert =
simpleGapInsert(checkRoute,i,newJob,adjJobEarly,newJobPi,2);
                        if(statusInsert >= 0){
                              cout << "I just placed job " << newJob-
>idx << " on route " << rteIdx << " after leg " << i << " using
simpleGapCheck." << endl;
                        }
                        return statusInsert;
                  }
            }//if(checkRoute.routeStatus == 1)
```

```
	}

	return -1;
}
int simpleGapInsert(route *jamRoute,int afterJobIdx,job *newGuy,float
newGuyStart, float newGuyPi,float loadTime){
// Called by simpleGapCheck if an appropriate gap is found.  This
routine will insert the job into the route
// and return a 1 if everything is successful, and a 0 otherwise
	int i,jobSlot;

	jobSlot = afterJobIdx + 1;
	if(jamRoute->numRouteJobs<MAX_ROUTE_JOBS){
		cout << "Num Route Jobs: " << jamRoute->numRouteJobs <<
endl;
		for(i=jamRoute->numRouteJobs;i>afterJobIdx;i--){
			jamRoute->JobList[i]=jamRoute->JobList[i-1];

		}
		cout << "i: " << i << endl;
		cout << "jobSlot: " << jobSlot << endl;
		jamRoute->numRouteJobs++;
		jamRoute->JobList[jobSlot].rj_i = newGuy->idx;
		jamRoute->JobList[jobSlot].rj_start = newGuyStart;
		jamRoute->JobList[jobSlot].rj_del = jamRoute-
>JobList[jobSlot].rj_start + newGuy->Cj;
		jamRoute->JobList[jobSlot].rj_ret = jamRoute-
>JobList[jobSlot].rj_del + newGuy->Rj + loadTime;
		jamRoute->JobList[jobSlot].rj_mile = newGuy->Mj;
		jamRoute->JobList[jobSlot].resched_ind = 0;
		jamRoute->JobList[jobSlot].rj_pt = newGuyPi;

		return afterJobIdx;
	}
	else{
		return -1;
	}
}
void main(){
	char inputFile[80];
	char outputFile[80];
	ofstream out;
	long runtime;
	clock_t begin,end;
	int diff;
	float secers;
	int j;
	int probSizes[6] = {20,40,60,80,100,150};
	int probGroup;
	int numProblems;
	int run_type;
	int lbJobOn,lbMiles,lbHours;

	RunSetup.maxMiles = 5500;
	RunSetup.HPW = 168;
	RunSetup.TA = 4;
	RunSetup.UT = 0;
```

```
	RunSetup.LT = 2;
	RunSetup.numIters = 1000;
	RunSetup.maxRtePHrs = 164;
	RunSetup.mph = 55;
	RunSetup.maxLoadSize = 42500;
	RunSetup.selCrit = 2;

	run_type = 1;

	cout << "Select run type (0: JobData2 (real problem), 1: Batch
Experiments, 2: Run Single Test Problem, 3: Evaluate Lower Bounds ";
	cin >> run_type;

	switch(run_type){
		case 0:
			//Run the real problem in "JobData2.txt" (no
backhauls)
			sprintf(inputFile,"ext/JobData2.txt");
			grabInput(0,&InputSchedule,inputFile);

			cout << "Filename: " << inputFile << endl;

			fixedNoBH(&InputSchedule,&RouteSchedule,&RunSetup);


	vomitRoutesAndInputSched(&RouteSchedule,&InputSchedule,"ext/compa
re.txt");
			vomitRoutes(&RouteSchedule);

			InputSchedule.inp_jobs[64].resched_ind = 1;
			InputSchedule.inp_jobs[30].resched_ind = 1;
			InputSchedule.inp_jobs[0].resched_ind = 1;
			insertJobs(&InputSchedule,&RouteSchedule,&RunSetup);


	vomitRoutesAndInputSched(&RouteSchedule,&InputSchedule,"ext/compa
re_post.txt");

			cout << endl;
			cout << "I scheduled all of those jobs on " <<
RouteSchedule.numRoutes << " routes" << endl;
			break;
		case 1:
			//Run a particular group of test problems (including
backhauls)
			cout << endl;
			cout << "Select a job size (20,40,60,80,100, or 150):
";
			cin >> probGroup;

			cout << "How many problems would you like to solve
(1-1000)? ";
			cin >> numProblems;
			cout << endl;

			cout << "Solving " << probGroup << " job size
problems." << endl;
```

```
                  grabBHMileData(&BHData);
                  for(j=1;j<=numProblems;j++){

      sprintf(inputFile,"../probs/%d/%d_%04d.txt",probGroup,probGroup,j
);

      sprintf(outputFile,"../solutions/%d/%d_%04d.txt",probGroup,probGr
oup,j);

                        grabInput(1,&InputSchedule,inputFile);

                        RunSetup.maxMiles = 5500;
                        RunSetup.HPW = 168;
                        RunSetup.TA = 4;
                        RunSetup.UT = 0;
                        RunSetup.LT = 2;
                        RunSetup.numIters = 1000;
                        RunSetup.maxRtePHrs = 168;
                        RunSetup.mph = 55;
                        RunSetup.maxLoadSize = 42500;
                        RunSetup.selCrit = 2;


      fixedBHLite(&InputSchedule,&RunSetup,&BHData,j,outputFile);
                  }

                  runtime = clock();
                  cout << "Well after " << runtime/CLOCKS_PER_SEC << "
seconds of execution time, I'm spent.  You can find my results in the
solutions folder." << endl;
                  break;
            case 2:
                  // Run a particular test problem (No backhauls)
                  cout << endl;
                  cout << "Select a job size (20,40,60,80,100, or 150):
";
                  cin >> probGroup;

                  cout << "Which problem would you like to solve (1-
1000)? ";
                  cin >> numProblems;
                  cout << endl;

                  RunSetup.maxMiles = 5500;
                  RunSetup.HPW = 168;
                  RunSetup.TA = 4;
                  RunSetup.UT = 0;
                  RunSetup.LT = 2;
                  RunSetup.numIters = 1000;
                  RunSetup.maxRtePHrs = 168;
                  RunSetup.mph = 55;
                  RunSetup.maxLoadSize = 42500;
                  RunSetup.selCrit = 1;
```

```
      sprintf(inputFile,"../probs/%d/%d_%04d.txt",probGroup,probGroup,n
umProblems);

      sprintf(outputFile,"../fullsolns/%d_%04d.txt",probGroup,numProble
ms);

                  grabInput(1,&InputSchedule,inputFile);

                  begin = clock();
                  fixedNoBH(&InputSchedule,&RouteSchedule,&RunSetup);
                  end = clock();

                  diff = (int)end - (int)begin;
                  secers = (float)diff / (float)CLOCKS_PER_SEC;

                  //output schedule to delimited file

      vomitRoutesAndInputSched(&RouteSchedule,&InputSchedule,outputFile
);

                  cout << "Number of routes: " <<
RouteSchedule.numRoutes << endl;
                  break;
            case 3:
                  //Evaluate the LBs for a particular group of problems

                  cout << endl;
                  cout << "Select a job size (20,40,60,80,100, or 150):
";
                  cin >> probGroup;

                  cout << "How many problems would you like to evaluate
LB for (1-1000)? ";
                  cin >> numProblems;
                  cout << endl;

                  cout << "Solving " << probGroup << " job size
problems." << endl;


      sprintf(outputFile,"../solutions/%d_lb.txt",probGroup);

                  out.open(outputFile);
                  for(j=1;j<=numProblems;j++){

      sprintf(inputFile,"../probs/%d/%d_%04d.txt",probGroup,probGroup,j
);

                        grabInput(1,&InputSchedule,inputFile);

                        RunSetup.maxMiles = 5500;
                        RunSetup.HPW = 168;
                        RunSetup.TA = 4;
                        RunSetup.UT = 0;
                        RunSetup.LT = 2;
                        RunSetup.numIters = 1000;
```

```
                        RunSetup.maxRtePHrs = 168;
                        RunSetup.mph = 55;
                        RunSetup.maxLoadSize = 42500;
                        RunSetup.selCrit = 2;

                        lbJobOn = jobOnLB(InputSchedule.inp_jobs,
InputSchedule.numSchedJobs,&RunSetup);
                        lbMiles = milesLB(InputSchedule.inp_jobs,
InputSchedule.numSchedJobs, RunSetup.maxMiles);
                        lbHours = hoursLB(InputSchedule.inp_jobs,
InputSchedule.numSchedJobs, &RunSetup);
                        out << j << "\t" << lbMiles << "\t" << lbHours
<< "\t" << lbJobOn << endl;
                  }

                  out.close();
                  runtime = clock();
                  cout << "Well after " << runtime/CLOCKS_PER_SEC << "
seconds of execution time, I'm spent.  You can find my results in the
solutions folder." << endl;
                  break;
      }//end switch(run_type)
}
```

gfuncs.h

```
template <class type1> type1 getMax(
      type1 val1,       // val1 to be compared
      type1 val2)       // val2 to be compared
{
      if(val1 >= val2){
            return val1;
      }
      else{
            return val2;
      }
}

template <class type1> type1 getMin(
      type1 val1,       // val1 to be compared
      type1 val2)       // val2 to be compared
{
      if(val1 <= val2){
            return val1;
      }
      else{
            return val2;
      }
}

template <class type1, class type2> type1 leftovers(
      type1 val1,       //find remainder of val1
      type2 val2)       //divided by val2
{
      register int howManyTimes;

      if(val1 < val2){
            return val1;
```

```
	}
	else{
		howManyTimes = (int) val1/val2;
		return val1 - howManyTimes*val2;
	}
}
template <class type1> int tceil(
	type1 raiseMe)    //value to be lifted
{
	if((raiseMe - (int) raiseMe) > 0 && raiseMe >= 0){
		return (int) (raiseMe + 1);
	}
	else{
		return (int) raiseMe;
	}
}
```

jobtypes.h

```
#define MAX_JOBS 150
#define MAX_ROUTE_JOBS 10
#define MAX_ROUTES 150
#define MAX_BH 100
#define MAX_LEG_BH 3

#define NUM_BH_SITES 32
#define NUM_FDC 41

struct job {
	int fdcIdx;
	int idx;
	int Mj;

	int chng_ind;
	float Ej;
	float Dj;
	float Lj;

	float Cj;
	float Rj;

	int ex_ind;
	float ExAj;
	float ExBj;

	int resched_ind;  //1 if job is to be inserted/rescheduled,
	//0 otherwise
};

struct backhaul {
   int bhID;
	int locID;
	float loadSize;
};

struct route_bh {
```

```
	int backhaulID;
	int locationID;
	int milesToPickUp;
	int milesToNDC;
	float bhStart;
	float bhCt;
	float bhRt;
	float loadSize;
};

struct route_job {
	int rj_i;
	float rj_start;
	float rj_del;
	float rj_ret;
	float rj_mile;
	float rj_legmt;
	float rj_pt;
	float rj_legpt;

	job job_props;
	int resched_ind;  //1 if job is to be rescheduled, 0 otherwise

	int numLegBH;
	struct route_bh BHList[MAX_LEG_BH];
};
struct route {
	int numRouteJobs;
	struct route_job JobList[MAX_ROUTE_JOBS];

	int routeStatus;  //1: route is active, 0: route is broken;
};
struct bh_sched {
	int numBH;
	struct backhaul inp_bh[MAX_BH];
} BHSchedule;
struct inp_sched {
	int numSchedJobs;
	struct job inp_jobs[MAX_JOBS];

	float mExAj;
	float mExBj;
} InputSchedule;
struct route_sched {
	int numRoutes;
	struct route bestRoutes[MAX_ROUTES];
	long bestIter;

	int lowerBound;
} RouteSchedule;
struct run_settings {
	float maxMiles;
	float HPW;
	float TA;
	float UT;
	float LT;
	long numIters;
```

```
      float maxRtePHrs;
      float mph;
      float maxLoadSize;
      int selCrit;
} RunSetup;
struct bh_data{
      int BH2BH[NUM_BH_SITES][NUM_BH_SITES];
      int FDC2BH[NUM_FDC][NUM_BH_SITES];
      int BH2NDC[NUM_BH_SITES];
} BHData;
```