1

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Rural School Bus Routing and Scheduling

by

Duncan John Forfar Chaundy

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

August 2, 1990

© Copyright

1990, Duncan John Forfar Chaundy

of Canada

du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

ISBN 0-315-60479-4

The undersigned recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis
Rural School Bus Routing and Scheduling
submitted by
Duncan John Forfar Chaundy
in partial fulfilment of the requirements for
the degree of Master of Computer Science

F. Fiala

Thesis Supervisor

M. D. Atkinson

Director, School of Computer Science

Carleton University

September 4, 1990

Abstract

The rural school bus routing and scheduling problem is an example of the vehicle routing problem. Given are some stops where students are waiting to be picked up, along with a network of roads where topographical features mean that not all neighbouring nodes are connected and a fleet of school buses. There are also time restrictions on travelling. This thesis presents a heuristic approach to solving this problem, generating feasible routes to deliver the students to the school using the smallest number of buses possible. Features of the algorithm include allowing more than one bus to serve a stop, allowing buses to run two routes for one school and allowing different bus speeds in different parts of the network. This algorithm is implemented along with a graphically interactive user-interface which allows the user to see the network, and to see an animated display of the resulting routes.

Acknowledgements

First I would like to most gratefully thank my supervisor Professor František Fiala for his supervision of this thesis. I would also like to thank Professor Bill Cunningham of the Department of Mathematics and Statistics for suggestions on future shortest path tree algorithms. Finally I would like to thank those fellow graduate students who shared their experience with Smalltalk and those who made helpful suggestions for improvements to the interface.

Contents

1	Introduction	1
1.1	Vehicle Routing and Scheduling	1
1.2	Problem Classification	2
1.3	Traveling Salesman Problem	5
1.4	Heuristic Approaches to Traveling Salesman	6
1.5	Classification of Solution Strategies in Vehicle Routing	11
1.6	Description of Some Routing and Scheduling Problems	12
1.7	School Bus Routing and Scheduling and Algorithms	14
1.7.1	Newton and Thomas	15
1.7.2	Angel, Caudle, Noonan and Whinston	18
1.7.3	Bennett and Gazis	18
1.7.4	Bodin and Berman	19
1.7.5	School Bus Scheduling	21
1.7.6	Savelsbergh	22
1.8	Lindenberg and Fiala	24
1.9	Complexity	27
1.10	Objectives	28

2	School Bus Problem	30
2.1	Data and Assumptions	30
2.2	Algorithm	34
2.2.1	Shortest Path Tree	35
2.2.2	Lower and Upper Bounds	37
2.2.3	All Pairs Shortest Paths	38
2.2.4	Route Generation	40
2.2.5	Furthest Branching Point	41
2.2.6	Isolated Fork	42
2.2.7	Knapsack Problem	42
2.2.8	Detour routes	45
2.2.9	Inner District Routes	48
2.3	Measures of Quality	49
2.4	Algorithm in Summary	51
3	Implementation and User Interface	53
3.1	Data Input	53
3.2	Network Display	57
3.3	Network Editing	62
3.4	Algorithm Execution	65
3.5	Route Display	70
3.6	Data Output	74
4	Computational Experience	76
4.1	Gerolstein Problem	76
4.2	Lindenberg and Fiala	78
4.3	St. Mark High School	79
5	Summary	82

6 Future Work	85
Bibliography	90
A Code for Routing and Scheduling	94
B Input Data and Timetables for Gerolstein Problem	107
B.1 Gerolstein Timetables	111

List of Tables

1.1	Classification of Routing and Scheduling Problems	4
1.2	Classification of Routing and Scheduling Problems – continued	5

List of Figures

1.1	Solution to Symmetric Traveling Salesman Problem	6
1.2	Initial tour and feasible 2-exchange	10
2.1	Isolated Components of the Network	38
2.2	Isolated Forks	42
2.3	Direct and Detour Routes	48
3.1	Network Display	57
3.2	Network Display and 'Map' Menu	58
3.3	Network with Distances, Node Demands and Node Names	59
3.4	Network with Rectangle for Magnification	60
3.5	Magnified Network	61
3.6	Network after User has selected to Remove a Node	63
3.7	'Execute' Menu	65
3.8	Shortest Path Tree with Travel Times	66
3.9	Window with Results of Heuristic Algorithm Execution	67
3.10	Window with Bus Statistics	68
3.11	Window with Arrival Inconvenience	69
3.12	'Clock' Menu	71
3.13	Network Animation in Progress	72
3.14	Network Animation with all Students at the School	73
3.15	Generated Timetables	74

Chapter 1

Introduction

1.1 Vehicle Routing and Scheduling

The vehicle routing and scheduling problem arises because an organization wishes to distribute some goods or people to a number of locations using a fleet of vehicles and the associated crews, and wishes to do this in an optimal way. The general problem has spawned many particular instances of the problem, not least because distribution costs add hundreds of billions of dollars to the price of goods alone [8]. For example, the annual distribution costs for a large distributor of propane gas in the U.S.A. was \$6 million in 1979. In 1974, the state of New York had a budget of between \$150 and \$200 million annually for school bus transportation. This budget was supplemented by funds from local school districts. The annual cost of leasing a school bus for 6 hours per day was approximately \$11 000 in 1974, rising to approximately \$20 000 in 1981. Also from [12] 18 806 students were transported on 848 school bus routes with a total daily distance of 56 142 kilometres. The total expenditure in 1988 was over \$11.5 million for just one school board.

The main output of a routing and scheduling problem is to provide a route and a schedule for each vehicle or for each driver. A route specifies the sequence of stops to be visited and the schedule specifies the time at which an activity should be

carried out at each stop. Thus there is a complete description of what tasks should be carried out, at what location and at what time.

Useful reviews of the subject of vehicle routing and scheduling appear in [8], [20] and [19]. Routing and scheduling is a much studied topic. The authors in [8] give nearly 700 references to works on routing and scheduling.

1.2 Problem Classification

Vehicle routing and scheduling problems can be separated into three broad categories. These are the routing problem, the scheduling problem and the combined routing and scheduling problem. In each of these problems there is a set of points where a vehicle must make a delivery or pick-up. In the routing problem, the vehicle must service each of these points but there are no time restrictions on when the points must be serviced. The solution to the problem gives an ordered series of points to be served by the vehicle. In the scheduling problem the time to service each node is fixed in advance and the solution to the problem gives a set of nodes to be serviced by the vehicle. In the combined routing and scheduling problem there are some time restrictions on when the points can be serviced or certain points must be serviced before others. The solution to the combined problem is to find the sequence of points and the time that each point is serviced by the vehicle.

One example is the case of newspaper delivery. Here we have some goods to be delivered to a number of locations using a fleet of vehicles based at a single depot. There are no *a priori* restrictions on the delivery times assuming that all the goods can be delivered in a relatively short period of time. Then the problem becomes a pure vehicle routing problem.

If there are time restrictions on when a location can be serviced then we have a routing and scheduling problem. In the dial-a-ride problem, the user must be picked up at one location and delivered to another location. The passenger must be picked up before being dropped off so there is a *precedence relationship* between these two stops. If the starting and ending time when a location can be serviced

is specified exactly then the location is said to have *definitive time for service*. A less restrictive constraint is that service must be within a *time window*. A *two-sided window* $[s, t]$ specifies the time that service may start and when it must end, for example a warehouse may only accept deliveries between 9:00 a.m. and 4:00 p.m., giving a time window of $[9:00, 16:00]$. In a *one-sided window*, either the starting time or the ending time is unspecified, so that service must either finish by a certain time or start after a certain time. This situation can occur in a dial-a-ride system.

Routing and scheduling problems can also be classified using the characteristics shown in tables 1.1 and 1.2. This table is based on classifications from [8] and [7].

Characteristics	Possible Options
Size of Vehicle Fleet	One vehicle Multiple vehicles
Type of Vehicle Fleet	Homogeneous (all vehicles are the same type) Heterogeneous (vehicles are different) Special types (e.g. compartmentalized)
Number of Depots	One depot Multiple depot
Demand Type	Known demands Probabilistic demands Partial satisfaction of demands allowed
Demand Location	Mostly at nodes Mostly on arcs Mixed
Underlying Network	Undirected Directed Mixed Euclidean
Vehicle Capacity Constraints	Imposed and all the same Imposed and different Not imposed
Maximum Route Times	Imposed and all the same Imposed and different Not imposed
Vehicle Operations	Pick-ups only Drop-offs or deliveries only Mixed pick-ups and drop-offs Split deliveries allowed or disallowed

Table 1.1: Classification of Routing and Scheduling Problems

Characteristics	Possible Options
Costs	Variable or routing costs Fixed operating or vehicle acquisition costs Common carrier costs for unserved demands
Objectives	Minimize total routing costs Minimize sum of fixed and variable costs Minimize number of vehicles required Maximize service or convenience Maximize customer priorities

Table 1.2: Classification of Routing and Scheduling Problems – continued

One example of a problem using this classification is the traveling salesman problem. It has a single vehicle with unlimited capacity housed at one depot, known demands on the nodes of an undirected network, with no restriction on the time of the route, and an objective of minimizing the cost which is the total distance traveled.

1.3 Traveling Salesman Problem

The traveling salesman problem is based on the problem of a salesman starting at his home city and traveling to each city on his list exactly once and returning home, and doing this so that the distance traveled in the tour is as small as possible. The traveling salesman problem may be formulated mathematically as follows [8]: Let a network $G = [N, A, C]$ be defined with N the set of nodes, A the set of branches or arcs, and $C = [c_{ij}]$ the matrix of costs. The cost of moving from node i to node j (the distance) is c_{ij} . The traveling salesman problem requires finding the Hamiltonian cycle in G of minimal total cost — a Hamiltonian cycle is a cycle passing through each node $i \in N$ exactly once. An example of a feasible solution to the symmetric — costs are equal in both directions along the edge — traveling salesman problem is shown in figure 1.1.

Figure 1.1: Solution to Symmetric Traveling Salesman Problem

As with most vehicle routing type problems, approaches to the traveling salesman problem are either optimal using mathematical programming or approximate using heuristic procedures. An assignment formulation of the problem appears in [8]. It is assumed that the costs are symmetric so $c_{ij} = c_{ji}$ and $c_{ii} = +\infty$ for $i = 1, 2, \dots, n$. The objective is to form a tour over all the nodes starting and ending at node 1, with the minimum tour cost. The decision variables are

$$x_{ij} = \begin{cases} 1, & \text{if arc } i - j \text{ is in the tour} \\ 0, & \text{otherwise.} \end{cases}$$

The solution to the assignment problem gives a matrix $X = (x_{ij})$ of decision variables so that exactly one arc enters each node and exactly one arc leaves each node, corresponding to a tour of the nodes. The objective function is to

$$\text{Minimize } \sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij}$$

There are further restrictions imposed so that the solution corresponds to one tour of the nodes and not two or more unconnected subtours.

1.4 Heuristic Approaches to Traveling Salesman

There are three main types of heuristic approaches to the traveling salesman problem. The first is *tour construction* where an approximately optimal tour is generated from the distance matrix. The second is *tour improvement* where improvements are attempted to an initial tour. The third is *composite* where tours are constructed by a tour construction method and then improvements are attempted.

One tour construction procedure is called the nearest neighbour [36]. This procedure constructs a tour by starting at a node, moving to its nearest neighbour, then moving to the nearest neighbour which has not yet been visited, of the last node visited. This process is repeated until all nodes have been visited. The last step is to return to the starting node, thus completing the tour. The worst case behaviour of the algorithm is

$$\frac{\text{length of nearest neighbour tour}}{\text{length of optimal tour}} \leq \frac{\lceil \log_2 n \rceil}{2} + \frac{1}{2}$$

where n is the number of nodes. The algorithm runs in time $O(n^2)$.

The Clarke and Wright Savings algorithm ([14] and [18]) again starts with a node as the central depot called node 1. Initially $n - 1$ subtours are formed, one for each node where each subtour consists of starting with node 1, visiting a node and returning to node 1. Then calculate the savings $s_{ij} = c_{1i} + c_{1j} - c_{ij}$ for $i, j = 2, 3, \dots, n$ and order the savings from largest to smallest. Starting with the largest savings the existing subtours are extended by combining subtours and so forming larger subtours by linking nodes i and j . Repeat until a complete tour is formed. The worst case behaviour for the cost of the tour is linear in $\log n$. The computation requires time $O(n^2 \log n)$.

Insertion procedures are also reviewed in [36]. An insertion procedure starts with a subtour of k nodes at iteration k and finds the next node to insert in the subtour and where it should be inserted. The nearest insertion starts with a subgraph consisting of only node i . Find node k so that c_{ik} is minimal and then form the subtour $i-k-i$. Then *select* node k not on the subtour closest to any node in the subtour. Then *insert* node k in the subtour by finding the arc (i, j) in the subtour which minimizes $c_{ik} + c_{kj} - c_{ij}$ and insert the node between nodes i and j . Repeat the selection and insertion steps until there is a Hamiltonian cycle. In the worst case behaviour of the algorithm, the length of the nearest insertion tour is twice the length of the optimal tour. The nearest insertion algorithm requires $O(n^2)$ computations.

The cheapest insertion procedure is the same as the nearest insertion except the selection and insertion steps are replaced by a combined one. Find arc (i, j) in the

subtour and node k not in the subtour, so that $c_{it} + c_{kj} - c_{ij}$ is minimal, and then insert node k between i and j . This requires $O(n^2 \log n)$ computational steps.

In the arbitrary insertion, the node to be inserted is selected arbitrarily. The worst case behaviour worsens to the tour being within $\lceil \log n \rceil + 1$ of the optimal, though the execution time improves to $O(n^2)$.

In the farthest insertion the selection step is changed to find the node farthest from any node already in the tour. Also choose the initial k by finding the maximal value of c_{ik} .

In the nearest addition procedure, first pick any node as the starting circuit with one node and zero edges. Then given a k -node circuit T_k , find the node z_k not on T_k that is closest to a node, y_k , on T_k . Let T_{k+1} be the $(k+1)$ -node circuit obtained by inserting z_k immediately in front of y_k in T_k . Repeat the selection and insertion steps until a Hamiltonian circuit is formed.

There are also convex hull algorithms. If the costs c_{ij} represent Euclidean distances and given the convex hull of the nodes in two-dimensional space, then the order in which the nodes that lie on the boundary of the convex hull appear in the optimal tour, follows the same order that they appear in the convex hull. For more details see [15], [21] and [34].

There are also approaches based on a minimal spanning tree. Both assume that the triangle inequality holds, i.e.

$$c_{ik} \leq c_{ij} + c_{jk} \quad \forall i, j, k \in N$$

where N is the set of nodes. The first is due to Kim [25]. First find the minimal spanning tree T of the graph G . The minimal spanning tree on the nodes of G is a connected graph on the same nodes which minimizes the sum of the edge costs. Then double the edges in the minimal spanning tree so as to obtain an Eulerian multigraph. Informally, a multigraph is a graph where an edge can occur more than once. It is Eulerian if each vertex has even degree. From this multigraph we can find a closed walk, that is a sequence of nodes where the first and last nodes are the same, and there is an edge in the graph between each pair of nodes in the walk.

This walk visits each edge of the multigraph at most once and each node at least once. Then using the triangle inequality remove edges from the walk where nodes have a degree greater than two, and add new edges which bypass nodes already visited in an enumeration of the nodes of the cycle. This transforms the walk into a Hamiltonian cycle. This procedure also gives a tour which is no worse than twice the optimal tour in cost. It runs in time $O(n^2)$.

Another minimal spanning tree approach is by Christofides ([13]). First find the minimal spanning tree T . In T , find the odd degree nodes, and solve a minimum cost perfect matching with these nodes using the original cost matrix. Add these edges to T to obtain an Eulerian multigraph where every node has even degree. Again form an Eulerian walk and remove edges from the walk where nodes have a degree greater than two to obtain a Hamiltonian cycle. This gives a worst case behaviour of

$$\frac{\text{length of Christofides' tour}}{\text{length of optimal tour}} \leq \frac{3}{2}$$

The execution time is limited by the time to perform the minimum matching which is $O(n^3)$, though usually the number of nodes with odd degree in the minimal spanning tree is significantly less than n .

Branch exchange heuristics are tour improvement procedures, introduced by Lin ([28]) and furthered by Lin and Kernighan ([29]). The branch exchange heuristics were initially 2-opt and 3-opt procedures then extended to the k -opt procedure. First find an initial tour which is generally chosen randomly. Then improve the tour using branch exchange heuristics until no more improvements can be made. This procedure terminates at a local optimum. A k -exchange of a tour is defined as the deletion of k branches in a tour and their replacement by k other branches forming a new tour. A tour is k -optimal if it is not possible to improve the tour using a k -exchange. The tour will generally terminate at a better local minimum the larger the value of k that is used, so that an n -opt tour is an optimal tour to the original problem. Figure 1.2 shows a 2-exchange. Note that in the example the costs must be symmetric since the direction on arcs (B,C), (C,D) and (D,E) is reversed.

Figure 1.2: Initial tour and feasible 2-exchange

A 3-opt procedure needs approximately n times the execution time that a 2-opt procedure needs. To get a tour which is close to optimal using 3-opt exchange, one should repeat the procedure for a number of starting tours and then choose the tour with the best result [36]. This means that the overall execution time can get fairly long, so instead composite procedures are used which give the same accuracy as repeated 3-opt procedures but at lower cost.

Composite procedures start by obtaining an initial tour using one of the tour construction procedures. Then use the 2-opt procedure on this tour, and the 3-opt procedure on the 2-opt result. This gives good results using an initially good solution and hopefully some improvement to near-optimal using the branch exchange heuristics. Variants on the composite procedure include bypassing the 2-opt procedure, bypassing the 3-opt procedure, running the composite procedure several times using different tour construction procedures, and running the tour construction procedure several times using different starting nodes and then continuing with just the best tour.

These traveling salesman heuristic procedures are for the symmetric traveling salesman problem. Some algorithms have been designed specifically for the asymmetric traveling salesman problem such as that by Akl [2] which is similar to that by Christofides. It uses a minimal directed spanning graph which is a subgraph spanning n nodes of a complete directed graph with a weight w_{ij} on each arc. It also has minimum weight and forms a tree when directions are ignored. In other

words, when the directions on the arcs are ignored, the graph is a minimal spanning tree with edge costs $c_{ij} = \min\{w_{ij}, w_{ji}\}$.

1.5 Classification of Solution Strategies in Vehicle Routing

In the vehicle routing problem a set of delivery routes are needed from a central depot to a number of nodes with demands. Each node has a service requirement. The objective is to minimize the total distance covered by the entire fleet. The vehicles in the fleet have capacities and may be limited to a maximum route time. All the vehicles start and finish at the central depot. There are several approaches to the problem and they can be classified as follows [8]: (1) cluster first-route second; (2) route first-cluster second; (3) savings/insertion; (4) improvement/exchange; (5) mathematical-programming-based; (6) interactive optimization; (7) exact procedures. Approaches (1) to (4) and (7) have been used in the past, and (5) and (6) have been developed recently.

Cluster first-route second procedures group or cluster nodes or edges together first and then form routes over the clusters as a second step. The route first-cluster second procedures work in the reverse order. First a large tour is constructed which includes all the demands though this tour is usually infeasible. Then the tour is partitioned into a number of smaller but feasible routes. This approach has been used by Newton and Thomas ([31,32,33]) and by Bodin and Berman ([6]) for the routing of school buses. These will be discussed in later sections.

The savings or insertion heuristics compares the current configuration with another configuration and chooses the one with either the largest savings in terms of some function such as total cost, or the one with the least cost for inserting a new demand entity. Both the current and the alternative configurations can be infeasible up to the final step when a feasible configuration is found. Examples of this approach are described in [14] and [34], and has been used for the routing of airplanes in [23]. Improvement and exchange heuristics include the branch exchange heuris-

tics developed by Lin and Kernighan ([28,29]) and mentioned above. Here a feasible solution is always maintained which is improved at each step with a reduced total cost, until no further improvements are possible. A modification of this approach has been used to schedule mini-buses for the subscriber dial-a-ride problem ([9]).

In interactive optimization, a high degree of human interaction is incorporated into the procedure for obtaining a solution. Since the human has experience in the problem he or she should be able to change parameters and evaluate the solution on subjective criteria. One advantage of this approach is that the system is more likely to be implemented and used. One example of this is [26].

1.6 Description of Some Routing and Scheduling Problems

In combined routing and scheduling problems, which often occur in applications, there are both task precedence relationships and time window constraints. Task precedence relationships mean that one activity must precede another, such as a pick-up occurring before a delivery and that these activities must be by the same vehicle. Time window constraints restrict the times at which these activities can occur. Some routing and scheduling problems are considered here besides the school bus problem. These are the tractor-trailer routing and scheduling problem, the routing and scheduling of street sweepers and household garbage collection vehicles, and dial-a-ride routing and scheduling.

The tractor-trailer routing and scheduling problem has two versions, one where the trailers have full loads and the other where the trailers have partial loads. The full-load problem involves tractors picking up a full trailer and transporting it directly to the delivery point. The load is delivered to just one point and the tractor can transport just one trailer. Thus a typical route involves leaving the garage, picking up a trailer, delivering a trailer, deadheading to get to another pick-up point, picking up another trailer, and delivering the trailer, and returning to the garage. Deadheading means that the tractor travels from one node to another without a

trailer. The demands in the problem are the number of trailers that have to be transported between a particular pair of nodes. There are then two decision problems to solve. The first is to minimize the total cost of handling all the demands, and the second is determine the optimal fleet size to service a subset of the demands, given that the remaining demands can be satisfied by a common carrier, i.e. the demands are satisfied by another fleet of vehicles. This second problem involves choosing which nodes should be part of the subset and which are the unprofitable demands which can best be serviced by a common carrier.

The application studied by Ball et al. [4] did not have severely restrictive time windows, but did have the following restrictions. (1) Certain pick-ups and deliveries had to occur at specific times of the day. (2) Routes served by a two-driver crew had a maximum of 120 hours of driving time and routes served by a single-driver crew had a maximum of 60 hours of driving time. (3) If there are multiple trips between the same pair of points, then these trips must be spread out over the week and not all scheduled for the same day.

In outline, the procedure to handle the problem starts by generating a single large route, assuming a single vehicle of unlimited capacity, which covers all the demands. This large route is then partitioned into smaller feasible routes which obey the workrule constraints and the time windows. For the second problem of finding a subset of nodes to be serviced, a route is constructed iteratively by inserting demand arcs from the demand node pairs and using the ratio of the change in the route cost to the change in the route time as a criterion as to whether an arc should be inserted or not.

Where the tractor loads are partial loads, a tractor may be carrying a load from more than one origin and heading to more than one destination at one time. Thus a typical route may differ from the one for full loads because there may be more than one pick-up or more than one delivery consecutively in the route. Otherwise the constraints on the problem are similar to those for the full-load problem.

The problems of scheduling street sweepers and household garbage collection vehicles are applications of the Chinese postman problem. This problem is an arc

routing problem since all the arcs in the network must be covered while minimizing the total distance travelled. Instead of considering the problem of visiting a number of sites on one street this is simplified to traversing one arc. In the street sweeper and garbage collection problems, a number of arcs need servicing and the problem is to minimize the number of vehicles needed. For both problems there are no precedence relationships between the arcs. For garbage collection there are no time windows either. In the street sweeper problem there are time windows since there are certain times when parking is banned on the streets and it is only in these time windows that the street sweeping can take place. It is only during these times that the street sweeper can have access to the curb. The basic approaches to the problems use either the route first-cluster second or cluster first-route second procedures.

In the dial-a-ride problem a customer calls a dispatcher requesting that the vehicle come and pick him or her up and deliver him or her to a destination. A time for pick-up or delivery may also be specified. There are two versions of the problem. In the dynamic or real time dial-a-ride problem the customers need immediate service so routing and scheduling is done in real time. In the subscriber or static dial-a-ride problem the customers call before they need service so that all the demands are known before the routing and scheduling is done. There are precedence relationships in both cases since the customer must be picked up before being dropped off. There may also be time windows since either the pick-up time or delivery time is specified and the time for the other service must be carried out within a certain interval of the specified time. These two-sided windows may be too difficult or impossible to handle so instead one-sided windows are used using only the one specified time.

1.7 School Bus Routing and Scheduling and Algorithms

In the school bus routing and scheduling problem as normally described in the literature, there is one or more schools, a number of bus stops with a number of students assigned to each stop, and a fixed time window for the pick-up and delivery

of the students. The main objective is to minimize the transportation costs which may be interpreted as minimizing the number of buses needed for a district, or as minimizing the number of vehicles and the operating costs. The data upon which the algorithms were tested was normally for densely populated suburban school districts. The authors in both [6] and [8] say that the only work they know of for the mixed loading rural problem is some unpublished work by the Santa Clara (California) County Center for Urban Analysis. (Mixed loading means that students from more than one school are on board simultaneously.)

1.7.1 Newton and Thomas

The works by Newton and Thomas [32,31,33] consider two school bus problems: the single school problem and the multi-school problem. In the single school problem we are given the location of a single school, the bus stops and the number of students to be picked up at each stop, a matrix of interstop travel times which is asymmetric because of one-way streets and restricted turns, the capacity of a bus, and the time it takes for students to board the bus. We are also given constraints which specify the maximum allowable time that students may ride on the bus, and whether or not students at one stop must be picked up by the same bus. It is assumed a bus cannot be overloaded, that all buses have the same capacity and that all the routes start and finish at the school. The objective is to find the minimum number of buses to transport the students to and from school and to find the time schedules.

The solution strategy is route first-cluster second. The initial tour considering just the stops is the solution to the traveling salesman problem using the nearest neighbour approach. This traveling salesman tour is then improved using a kind of 3-exchange. In order to ensure that no arcs are traversed in the reverse direction two consecutive arcs are replaced by one arc, and a single arc is replaced by two consecutive arcs where this exchange results in a lower cost tour. This process is repeated until no further improvement can be made. If the time matrix is symmetric then the directions can be reversed without increasing the tour cost, so then there is a further step of improving the tour by using 2-exchanges.

This single tour is then partitioned into feasible bus routes using the data and constraints from above. A bus route starts at the school and visits the bus stops in the order determined by the traveling salesman tour. The bus traversing the route continues to visit stops in tour order and picking up students until either the bus is full or the maximum riding time is exceeded for the first student. The route terminates by returning to the school. The next route starts at the school and visits the node following in the traveling salesman tour, the last node visited by the previous bus.

In the multi-school problem the authors extend the previous problem to consider a school district with several schools. The starting times of the schools will be staggered so that the buses can be used for more than one school. A bus only picks up students for one school in any one route. These students are then delivered to the school. If the bus makes a second trip then it is to pick up students for a school with a later starting time. Thus there are non-overlapping time periods during which a school opens. The problem here is to determine for each time period, which school a bus is assigned to and the stops it is to visit and the times to visit them. In addition to the information needed for the single school problem, the number of school-opening time periods, the initial location of each bus and the time period in which a school must be served are also needed. Here students at one stop must all be picked up by the same bus. Also the bus capacity can be specified for each school since this may vary depending on the size of the students. The objective is to minimize the total time for all routes in a given time period and to minimize the number of routes for a particular school.

The solution strategy fixes the number of routes needed for each school and then finds an assignment of stops to each route and their order that minimizes the total travel time for all routes in a given time period. The overall routing problem consists of sequential optimization from period to period. For each school in a time period, the first step is to determine the number of routes needed to service the school. For school S , a lower bound for the number of buses is given by:

$$R_{S,\min} = \left\lceil \frac{\text{number of students to be transported to } S}{\text{bus capacity}} \right\rceil$$

When this minimum cannot be attained because of the constraints, an allowance L , is made for the number of routes allowed, so that

$$R_{S,\max} = R_{S,\min} + L$$

The authors find that taking L to be one works best. If the following steps fail to give a solution, then L is increased by one and the steps repeated.

The second step is to determine a lower bound to the average route travel time from each origin to each school in the time period. This is found using the interstop travel time matrix and assuming that each route must have at least one stop. Then the origins of the bus routes are chosen. For the first time period the origins are the bus garages and for the others they are the destinations of the previous routes. It is assumed that if the maximum number of routes allowed are used then it is possible to visit all the nodes. Then for each school find the origin with the most routes and call it the super-origin. If N stops are allocated to a school, generate trial routes using all N nodes. Each of the $N - 2$ routes starts with the super-origin. Route i of the $N - 2$ routes has a second node of $i + 1$, where node 1 is the super-origin and node N is the school. The route continues using the nearest neighbour approach. Find the route with the least cost.

Each of the possible tours, not just the one of least cost, is then partitioned into routes using the same approach as for partitioning in the single school case. Each route starts at the super-origin. If the number of routes is within $R_{S,\max}$ then try to improve each route using exchange heuristics. Calculate the total travel time for the set of routes. Using all the initial tours, find the set of routes with the least total travel time. In addition the tour of least cost is also improved using exchange heuristics and this is also partitioned into routes. Then using the best set of routes found overall, allocate the actual routes using actual origins in such a way as to minimize the additional travel time.

1.7.2 Angel, Caudle, Noonan and Whinston

The authors of [3] have a similar objective of minimizing the number of routes while keeping the mileage to a minimum. As well buses should not be overloaded and there is a maximum time to traverse a route. The usual parameters are the numbers of each capacity bus, the loading time per student and the extra time needed per stop. Distance and travel time matrices are also needed. The time matrix is formed from the distance matrix and bus speeds on the links.

The routing and scheduling algorithm has two components. First the stops are clustered and then routes are formed from the clusters. The algorithm starts by making a separate route for each node. Then for each pair of existing routes, a measure of proximity, or association, is calculated for a savings type procedure. This measure is calculated using the nearest nodes of the routes, the distance from the school, the current load and the remaining time allowed for the route. Then routes are merged starting with the pair with the greatest measure. When two routes are merged a traveling salesman algorithm is used to find the shortest overall route. This process is continued until no further merging can be done.

1.7.3 Bennett and Gazis

The bus routing procedure in [5] is based on the Clarke and Wright savings method. They use this method to allow for asymmetries in the distance matrix, to allow for a separate origin and destination of each bus, and to allow for the safety and comfort of the students.

In the algorithm, first each bus stop is given its own route or routes if more than one bus is needed. The routes are assumed to start at a common garage and then proceed to the school. Pairs of routes are combined by forming a route starting at the garage, covering the bus-stop part of the two routes and going to the school. The savings function is similar to the Clarke and Wright function: the time from the last node of the first route to the school plus the time from the garage to the first node of the second route minus the time from the last node of the first route

to the first node of the second route. The savings are found for each ordered pair of nodes, (a, b) , and ordered by the amount of savings. Ordered pairs are used since the cost matrix is asymmetric. A pair of nodes is eligible when a is the last stop on a route and b is the first stop on a route. However the authors also consider other possibilities.

The first is that there may be increased savings if the order of the bus stops on one of the routes is reversed. Also in order to increase safety and convenience, the travel time should be kept low. For this purpose a modified objective function is used: $W = B + \alpha S$ where B is the total bus travel time, S is the total student travel time and α is a weighting factor. A modified savings takes into account the fact that students from the original first route must now travel on the second route before getting to school. The routes are combined as before but now the objective is to maximize at each step, the decrease in W . Once the routes are formed they are then improved using a 3-opt algorithm so that routes are 3-optimal relative to inversion and insertion. Inversion is the route produced by inverting all the arcs of the route. This then takes the asymmetry into account as in the work by Newton and Thomas.

1.7.4 Bodin and Berman

In [6] the objectives are again to minimize the transportation costs, the average transportation time per student and to generate timetables for the school district. The authors mention that where the transportation costs of a district are subsidized by another level of government, the aid formulas may have factors besides the number of students transported. This means that it may be more economical for the local district to have more buses than that implied by a strictly minimum cost solution. They assume that there is more than one school and that the times of classes starting and finishing is fixed. Also that these times can be partitioned into distinct time periods and that a bus only services one route in each period. Thus to minimize the capital cost, the problem becomes one of minimizing the maximum number of buses needed in any one time period. So no bus should be idle in the

peak time period, and in the off peak time periods routes can be shorter than in the optimal case. The main method to reduce the number of buses needed in the peak period is by adjusting the starting times of the schools. They first find the routes for each school and then adjust the starting times to minimize the maximum number of routes.

In order to facilitate the input of the data, the authors introduce the idea of a ministop. This is a location which could be used as a bus stop. A student is assigned to the ministop nearest his or her home and this ministop remains unchanged regardless of the actual stop used as, say, the student changes schools within the district. This allows the district to be treated as a single network and allows for flexibility in changing the network such as changing the numbers of students attending the schools and changing the locations of the schools.

The routing algorithm is of the route first-cluster second variety and similar to that of Newton and Thomas. The authors say that is appropriate for a densely populated area where the bus capacity is often reached before the maximum time allowed for the route is exceeded. However for a rural area they would have used a cluster first-route second procedure. To find the traveling salesman tour they use the 3-opt procedure. This tour is then partitioned into routes though in this case the routes are left open, that is, only the bus stops on the route and the school are considered. The bus also has to proceed from the end of the previous route to the start of the next, but for a suburban district this can be added to the route without exceeding the time allowed for a route. In addition they use a look-ahead feature so that if adding bus stop j_m to the route (school i , stop j_1 , stop j_2 , ..., stop j_{m-1}) would exceed the capacity of the bus then stops j_{m+1} and j_{m+2} are considered instead. The next route would then start with stop j_m . If the student load at stop j_m means that the bus capacity would be exceeded and the load at j_m was at least 10% of the bus capacity, then j_m is partitioned into two stops and the algorithm restarted.

Using these routes the best starting and ending times for the schools are determined manually so as to minimize the maximum number of buses needed.

The objective in the scheduling component of the algorithm is to minimize the total travel time of all buses. If the starting times of the schools fall into distinct time periods, then the problem becomes one of assigning the buses in consecutive time periods. For the first time period n buses can serve the n routes. For the remaining time periods, the buses have to be assigned to the routes knowing the positions of the buses in the previous period, the partial routes already constructed, the pick-up and drop-off times and the time to reach the first stop on the partial route. When the route is completed with the addition of the origin, the route is improved if necessary using the 2-opt procedure. Where possible the use of a new bus is avoided by trying a rerouting with less efficient routes for the time period in question.

1.7.5 School Bus Scheduling

The school bus scheduling problem is somewhat different from the ones above. The problem here is to find school bus schedules given the routes for a school and given certain time windows for the buses to arrive at the school. The objective again is to minimize the total travel cost among the feasible schedules which require the fewest buses. The problem arises because the city specifies the routes and the separate schedules. It is then up to a private company to perform the schedules.

In [22], the authors compare scheduling algorithms. They look at three heuristics. The first and second are due to Orloff [35]. For the first heuristic constructs an acyclic graph in which the nodes are the routes and the arcs represent feasible sequences of trips. A chain of nodes in the graph corresponds to a combination of routes. A chain with more than one arc may not correspond to a feasible combination if there are time windows to consider. The problem is then solved by finding the minimal chain decomposition of the graph with minimum cost. Any infeasible combinations must be adjusted manually.

The second heuristic uses the same graph as before except that now undirected edges are used. Each edge (i, j) is assigned a weight $w_{ij} = M - c_{ij}$ where c_{ij} is the travel cost on the edge, and M is large. A maximum weighted matching problem

is solved to give feasible combinations of length zero or one. The matched nodes are then collapsed into one node and the problem re-solved. This continues until no new combinations are formed, which is usually three or four matchings. If need be, the schedule can now be improved using a 3-opt procedure.

The third heuristic is that by Swersey and Ballard [38]. They consider the more general problem than assuming that the time periods are distinct, but they do use discretized time windows rather than continuous ones to make the problem tractable. The problem is then solved using an integer programming formulation. Manual adjustment may again be required.

The main conclusion seems to be that the first Orloff heuristic is the one which best balances the results with the computation time.

1.7.6 Savelsbergh

In [37], Savelsbergh considers the vehicle routing and scheduling problem with constraints of time windows at the customers, a mixture of pick-ups and deliveries and precedence relationships between customers. The solution to the problem is approached in a cluster first-route second manner though with an important contribution by human guidance. The system implementing these algorithms is an interactive planning system since it not only has algorithms incorporated into it, but supports planning activities using human perception through the use of a graphical and interactive user interface. He discusses the design and the approach to the user interface developed for CAR (Computer Aided Routing), and the reasons for the kind of interface chosen. It is a tool to enable an expert planner to construct economical vehicle routes and schedules; the user is the one to make the planning decisions.

The interactions of the system with the user are menu driven. There are three main menus: communication, clustering and routing. The communication menu allows the user to change the data and to write out partial solutions. Nodes can be added from a data file, a node can be deleted from the set of nodes, the data for a node can be changed and a trip can be fixed and written to a file. The clustering

menu has commands for the clustering phase of the algorithm. A cluster is a group of nodes with a vehicle to service them. The clusters are generated using seed points which are the points around which a cluster is grown. The seeds and clusters can either be generated automatically or they can be created and edited manually. The routing menu contains commands to form routes where a route has one or more trips, and a trip is a sequence of nodes starting and finishing at the depot. There are commands for the automatic generation of routes and trips, and the manual creation and editing of routes and trips.

In addition there are information, screen and storage menus. The information menu allows the user to examine information for the nodes, vehicles, clusters, trips and the overall plan. From the screen menu the user can see a spatial representation of the problem and the trips shown in separate colours, and zoom in on a particular region. The trips can be displayed as bar charts by time and vehicle loads, or by text. The storage menu allows the user to store and retrieve sets of trips.

When the seeds are selected automatically, they are chosen based on the difficulty degree of a customer, which is based on the distance from the depot, the size of the load and the narrowness of the time window. The nodes selected are then refined by selecting nodes which are geographically dispersed and which would have the greatest load if all loads in a cluster went to the seed and capacity constraints could be ignored. Then the vehicles must be allocated to the seeds. The customers are assigned to the seeds by considering the customer which has the maximum difference between the smallest and second smallest cost of a feasible assignment. The customer is assigned to the vehicle where the cost is least. Then improvements are tried by reassigning customers to another vehicle or by swapping two customers. The trips are constructed by solving a traveling salesman problem using an insertion heuristic subject to the time window constraints and the capacity constraints. These trips can then be improved by exchange heuristics using heuristics to find suitable nodes for exchange which do not violate the side constraints.

Savelsbergh also presents a system for defining an abstract vehicle routing problem. This is a formal language to describe how a problem fits into different categories

in an expanded version of tables 1.1 and 1.2. He also outlines a future model and algorithm management system, an expert system to analyse a problem, pick a model and suggest suitable algorithms for solving the problem.

1.8 Lindenberg and Fiala

The work which this thesis draws upon the most is that by Lindenberg and Fiala [30]. The algorithm described in chapter 2 is a modification of that described in [30]. The authors also implemented a version of the algorithm in Fortran IV though no use was made of this program apart from the data they used for the Gerolstein problem discussed in section 4.1. In order to avoid repetition the discussion of Lindenberg and Fiala's algorithm will be made with reference to the algorithm described in more detail in chapter 2.

The objective in [30] is to find the minimum number of buses from a given set of buses to transport the students to school, and to form the routes and timetables used by the buses. The explicit assumptions are that classes start at the same time for all students, there is a maximal riding time which is the same for each student, there is a maximal waiting time before classes start, that each location has exactly one bus stop, that the buses in the fleet can have different capacities and that a bus can start at any location at any time. Also they assume that there is only one school and that the demands are splittable. It is not made clear whether or not buses can pass through the school and not drop the students off. It is also assumed that every bus has the same average speed on every edge. The network is made up of the nodes, the school, the edges corresponding to the roads with the distance between the nodes and the student demands. The bus fleet is ordered by bus capacity.

The authors then introduce a number of concepts used in the algorithm. The first is the idea of the inner and outer districts, and the fact that the number of students in the outer district gives a lower bound to the number of buses needed. Having an inner district also means that closed inner routes are possible. The routes that the buses follow can be classified into those that are direct and those that have

detours. The detour routes are classified into type 1, type 2 and type 3. Type 1 routes correspond to the type 7 routes in chapter 2, where the first one or more nodes are not on a shortest path but the remaining ones are. Type 2 routes have the initial nodes on a shortest path but before reaching the school a detour is made while the remaining nodes are on another shortest path. Type 3 routes are a combination of type 1 and type 2 in that there is an initial detour, part of a direct route and again another detour before proceeding to the school. Examples are presented where each type of detour is used.

The shortest paths to the school are found from each node giving the shortest path tree, though the cost used on each edge is the distance and not the time which is used in the algorithm described in chapter 2. The nodes in the tree can then be classified into leaves, inner points and branching points. Branching points in the shortest path tree also imply that these nodes are the roots of subtrees called forks. If a fork contains another fork then it is a fork of higher than first degree. Forks can also be isolated if their branches are pairwise non-neighbouring. Two branches are neighbouring if there is an edge between a node in one branch and a node in the other. Also the shortest path tree can be split into subtrees and hence define isolated components of the network if there are no edges in the network between these subtrees.

The bus fleet is decomposed in such a way as to find the minimum number of buses needed to transport the outer-district students. However the buses are used in decreasing order by size until the last bus when the smallest bus needed to transport the remaining students is used. This means that if this smallest bus cannot be completely filled but a smaller bus could be used, then the number of buses needed must increase. This subset of buses used is called a representation of the students. The number of buses is the length of the representation. The number of empty seats of the buses is called the defect in the representation.

Then some solution strategies are presented. First the minimum number of buses (*Min*) can be found. This lower bound may have to be increased if the isolated components are considered separately. In order to keep the number of possible

solutions under control, look for some structures in the network which can help to provide a focus for the routing.

The first of these is the forks of the first degree and to find its isolated components. First find the bus representation of the component. Then solve the knapsack problem discussed in section 2.2.7. Find the corresponding route and a bus to serve it. If there is unused capacity fill the bus on the way to the school or by using a detour. When the isolated component has been fully served, the number of buses used can be compared to the minimal bus representation to see if Min will have to be exceeded. The non-isolated components of the fork can be separated into branches which are pairwise isolated and those which are not. It can also be important to ensure that a route starts with a boundary node, i.e. one lying on an outer branch. The distance between two branches of a fork can be defined as the minimum of the distances between nodes of each branch. The routes for non-isolated components is based on finding the minimal bus representation for the students in the fork. Then find the distances of the branches and if outer branches exist, they will be the branches with the largest distance. Then select the longest of the branches with the largest distance, B . The number of students on the branch is S_B . Then find the minimal bus representation of S_B using the buses from the representation of the fork. The defect in the representation is D_B . The goal is to eliminate the D_B by using detours or by filling seats on the way to school. Denoting the defect for the network by D , if D_B is greater than D then at least $D_B - D$ students need to be picked up outside the branch in order to avoid an increase in the number of buses needed.

Where the defects satisfy $0 < D_B \leq D$ then at least one bus will need to make a detour. Making detours to fill the bus will be tried first with type 1, then type 2 and finally with type 3 detours. In order to find where the students in the branch should be picked up, define a k -block defined by q , where k is the number of students that can be picked up in the block by the bus which cannot be filled in the branch, $k = c - D_B$, where c is the capacity of the bus. A k -block represents a sequence of consecutive nodes along a branch, containing at least k students. q is the node in

the k -block closest to the branching point. The position of q can be a determining factor in trying to find detours. Starting with q closest to the school, find possible type 1 detours which join the direct route at the most distant node of the k -block which can fill the bus in the remaining time. If the D_B students cannot be picked up using a type 1 detour then a type 2 is tried with the detour starting at q , with possible detours being constrained by the time remaining for a detour. If a detour still cannot be found then try a type 3 detour where there is a detour both before and after the k -block. If this also fails then the search for a detour starts again this time by moving the k -block away from the branching point. If detours still cannot be found after trying all possible positions for the k -block, then use the detour with the maximum number of students, and try to fill the bus on the way to the school. It could also be the case that a better route could be found by moving the k -block to another branch. Where the root of the fork is in the inner district then priority is given to nodes in the outer district. These processes continue until all the nodes in the outer district are served.

Then the students remaining in the inner district can be served with closed routes using buses from the outer district routes. This assumes that there are more students in the outer district than in the inner. If this is not the case then use more buses for the inner district. A closed route is started by visiting the nearest neighbour and then inserting more nodes so that the total route time and capacity are not exceeded. The bus for the closed route could be a bus from the outer routes or that bus could be replaced by a larger bus which was not used in the outer routes if the bus capacity would otherwise be insufficient. If a bus is reused then some students may have to be left behind in the inner route if the capacity is insufficient.

1.9 Complexity

It is important to note that almost all of the problems mentioned above are computationally hard. In the summary of NP-hardness results presented in [27], the vehicle routing problem, the traveling salesman problem, the mixed Chinese postman prob-

lem and the multiple depot Chinese postman problem are NP-hard. However the Chinese postman problem itself and the directed Chinese postman problem are in $\mathcal{P}$. The vehicle scheduling problem with fixed arrival times is in $\mathcal{P}$, but if there is a time window even if the same for all schedules then the problem becomes NP-hard. Savelsbergh shows in [37] that the traveling salesman problem with time windows (there is a time window during which the node can be visited) is NP-complete in the strong sense (that is that the result holds even with a unary encoding of the data). The proof starts from the traveling salesman decision problem which is known to be NP-complete in the strong sense ([17]). He also shows that the traveling salesman problem with mixed collections and deliveries where there is a load to be collected or delivered at each node with a fixed capacity vehicle, is NP-complete using the 3-partition problem which is NP-complete.

To see what the formulation for an exact solution would be, Newton ([31]) formulates the school bus problem as an integer program. The problem is for one school, 10 origins, 20 routes and 100 bus stops. The integer program would require 213 840 variables and more than $40(98)!$ constraints.

1.10 Objectives

The object of this thesis is to design and implement an algorithm to solve the rural school bus routing and scheduling problem, using as few buses as possible, and so that the algorithm generates timetables for all the students. As mentioned above, the problem is computationally hard. In order to make the solution method more tractable, the method used will be heuristic.

The objective function used in the problem is to minimize the number of buses which are needed to transport the students from their bus stops to the school. This simple objective function can be used because the cost of paying the drivers is often the major cost in the cost of transportation [8]. An alternative objective function is to minimize the number of passenger-minutes for transporting the students to school since this means that safety is maximized.

The implementation of the algorithm should run in a 'reasonable' time, which is taken here to mean a few seconds for problems with approximately 30 nodes and tens of minutes, not hours, for larger problems. The execution times are of secondary importance to obtaining a working implementation, especially considering that the language of implementation (Smalltalk) will give slow execution times. On the algorithm itself the main emphasis will be on the generation of the routes and timetables and not on preprocessing that may be needed beforehand.

An equally important or perhaps more important contribution to the work should be a user-friendly interface which allows the user to easily visualize the network of roads and the associated data. The user should also be able to easily adjust the network itself and the constraints and parameters so that the user can rerun the algorithm under different conditions. The interface should also allow the user to enter data graphically if desired, and to see the timetables graphically and be able to interact with the display of the timetables.

Chapter 2 contains a description of the problem, the kind of data needed for the problem, the assumptions that have been made and the constraints and parameters. There is also a description of the heuristic algorithm used to solve the problem. A more detailed high-level-code description appears in appendix A. Chapter 3 contains a description of and examples of the user interface, and how the user can interact with the program. Chapter 4 describes the results of using the program on different sets of data. Appendix B contains the data for one problem and the timetables that were generated for it. Chapter 5 contains a summary of the work. Chapter 6 has some suggestions for possible additions and improvements to the algorithm and its implementation.

Chapter 2

School Bus Problem

In the rural school bus routing and scheduling problem students waiting at bus stops must be picked up and taken to their school within the restrictions laid down by the school board. The objective is to transport the students to their school with the minimum number of buses.

2.1 Data and Assumptions

We are given a network, i.e. a set of nodes representing the bus stops and a set of edges representing the roads. There may be additional nodes representing road junctions. Those nodes which are road junctions are artificial in the sense that they do not correspond to actual bus stops, but in all other ways they are treated as the other nodes are. There are also nodes representing the schools in the network. We make the assumption that there is only one central school to which the students are to be delivered. We make this assumption primarily for simplification and because in real life there may only be one school or buses are only used for one school. There are demands on the nodes representing the students waiting at the bus stops. Those nodes which are road junctions and the school have a demand of zero. The edges are normally undirected, although in general they could be directed.

The rural nature of the rural school bus routing and scheduling problem arises because topographical features, such as rivers or mountains, in the area surrounding the school mean that a pair of nodes which are near neighbours in the network

modelling the area, may not have a road connecting them. Where there is a river there needs to be a bridge to cross it in real life and with mountains it may be impossible to build a road suitable for a school bus. This rural nature is in contrast to an urban or suburban area where the network is more grid-like. In addition in an urban or suburban area the students' homes are much closer together. It is then an important step of the algorithm to determine where the bus should actually stop to pick up the students. This would be either because students from more than one home are picked up together or because the bus may not need to travel down the road where the student's home is located. In the strictly rural problem by contrast, the stops are already fixed at some point in each village. The students at each village must assemble at this predetermined bus stop.

There is a cost associated with each edge representing the distance along the road between the stops. The edges are also split into disjoint sets where each edge set represents edges which are in the same sort of driving environment. In one of the examples to be presented in section 4.3, the edges are divided into two sets: village and rural.

To present this more formally, the network N is given by $N = (V, E)$ where V is the set of network nodes or vertices, and E is the set of edges. The network nodes are given by

$$V = \{v_1, v_2, \dots, v_n\} .$$

There is a demand corresponding to the number of students on each node where the number of students at v_i is s_i . The set of edges is given by

$$E = \{e_1, e_2, \dots, e_m\} ,$$

where $e_i = (v_j, v_k), j \neq k$, and the distance on the edge or edge cost on e_i is c_i . There are also disjoint subsets of E given by

$$E = F_1 \cup F_2 \cup \dots \cup F_q ,$$

$$F_i \cap F_j = \emptyset .$$

There is a fleet of buses provided to transport the students to the school. The capacities of the buses are also given and are fixed for each bus, although not all

buses need to have the same capacity. We assume that each bus can start at any node at any time. If there is a central depot for the school buses then there would have to be some overhead while the drivers drive to the start of the routes. The assumption is also reasonable if the buses are stationed at scattered locations and it is the responsibility of the bus driver to get to the start of the route at the correct time. Thus the bus fleet can be represented as

$$B = \{b_1, b_2, \dots, b_r\} .$$

The capacity or the maximum number of passengers for bus b_i is denoted by p_i . The average speed of the buses is given for each set of edges in the same driving environment. Having an average speed for each edge set means that each average speed can be varied independently, rather than having one average speed for the network and adjusting the actual distance along an edge by some factor to reflect the actual average speed for that edge. It also means that the cost on each edge in terms of time can be adjusted easily. The distances between the nodes in the network are fixed, but the time to travel a certain distance could vary depending on, for instance, the quality of the road surface and the time of day. This speed is assumed to be the same for every capacity bus. This may not be valid if buses vary greatly in size and if some buses in the model are in reality vans or cars. So for edge set F_i the average bus speed is a_{F_i} . The average speed a_{F_i} represents not only the actual speed of the buses but includes the time used up at the bus stops when stopping to pick up students.

Initially, the available bus fleet is given. The buses are kept in a sorted list in decreasing order of their capacities so that when a bus is required for a route, the largest bus can be retrieved in constant time. The number of buses can be minimized if the larger buses are used in preference to smaller ones.

The school board or other authorities are expected to specify certain time limits to constrain the problem. The first is the maximum time that the students can ride on a school bus. This is assumed to be the same for all students though if the school has students of widely differing ages then this may not be valid. However for most schools big enough to have school buses it is expected that the time limit will be

the same for all students. The school board also specifies the maximum time that students can wait at the school before classes start. This may be partly so that students do not have to wait around before classes start and partly so that they do not have to be supervised before formal supervision in classes starts. Again this maximum waiting time is assumed to be the same for all students. We are also given the time that classes start which again is assumed to be the same for all the students. These time restrictions thus give time windows within which the problem must be solved. These times are: T_{ride} , T_{wait} , T_{start} .

The objective is to minimize the number of buses needed from the fleet to transport the students to the school. The number of buses is a simple function by which to measure the cost of transporting the students since the cost of paying the drivers is often the major cost for the school board [8]. Minimizing the number of buses could be a realistic criterion for minimizing costs because it may be the case that union rules specify that a driver is to be paid for a four-hour morning shift regardless of whether or not the driver is actually on the road. Thus reducing the time that a driver works will not reduce the cost of paying the driver. In a private communication with F. Fiala, where a more complicated cost function is used, the cost in dollar terms is heavily influenced by the number of buses needed. An additional objective that could be used is to minimize the total number of passenger-minutes since for maximum safety it is desirable to minimize the time that students spend on the buses [6]. Thus the objective function is

$$g(x) = \min \sum_{i=1}^r x_i ,$$

where $x_i = 1$ if bus b_i is used to transport the students to school, $x_i = 0$ otherwise.

There are also two parameters to be specified for the model. The first is whether or not the demand at a node is splittable. That is can more than one bus pick up students at a stop. Multiple buses serving a stop may not be allowed because confusion may arise as to which bus which students are to board. The second parameter is whether or not one continuous route can have the school as an intermediate node, that is can the students remain on the bus even though the bus may reach the school

before all the students are finally dropped off. It may be felt that it is unfair to the students to ask them to stay on the bus even though they have already reached the school. In a rural network it may not be such a problem to ask students to remain on the bus past the school since when the bus appears to pass the school in the model, in real life the bus may only pass through the village where the school is located. Thus students do not actually see the school until they finally arrive there. The advantage of allowing students to remain on the bus is that the route can normally be longer than it would be otherwise. This is because normally $T_{\text{ride}} > T_{\text{wait}}$.

In the present work there is no particular allowance made for delays while students are picked up. These delays occur because the bus must stop and open the door to allow the students to board. So there could be a parameter specifying the number of seconds that it takes for one student to board the bus. A more sophisticated model may have a function of the number of students to be picked up giving the delay time. However in the present model such delays are incorporated into the average bus speeds.

It is also assumed that the timetables need be generated for the morning only as the afternoon timetables are simply the morning timetables run in reverse.

2.2 Algorithm

The problem is NP-hard (see section 1.9) and computationally intractable, so the present approach to solving the school bus problem is heuristic. It is based on the algorithm described in [30]. The following features explained below which are from the Lindenberg-Fiala model and algorithm are incorporated here: (1) the objective is to find the minimum number of buses; (2) T_{ride} , T_{wait} , and T_{start} ; (3) a heterogeneous bus fleet; (4) only one school; (5) inner and outer districts; (6) type 1 detours are similar to type 7 detours; (7) routes are based on the shortest path tree and the tree can be used to find branching points; (8) finding isolated components of the network; (9) finding lower bound to number of buses; (10) using isolated forks to solve a knapsack problem; (11) using closed routes in the inner district.

The following subsections describe in some detail the algorithm followed. (For more details in the algorithm, the interested reader is referred to the code shown in appendix A.) In two preprocessing steps, the shortest path tree rooted at the school and the shortest path between each pair of nodes in the network are determined. After the preprocessing, the routes and schedules are generated simultaneously. The results obtained can then be measured against an upper and a lower bound.

2.2.1 Shortest Path Tree

The first step is to find the shortest path tree where the school is the root of the tree and the costs on the edges are the times to traverse each edge. The travel time on an edge is calculated from the distance given for the edge and the average bus speed for the edge set containing the edge. Within this tree we can identify the leaves and the branching points. Branching points are those vertices which have degree greater than two in the tree. The remaining nodes are inner points of the shortest path tree.

The vertices of the tree are the vertices of the original network. Other work such as that by Bodin and Berman [6] has an initial step of clustering the actual vertices into *ministops* which will be used in the generation of routes. In the present implementation, this is not done despite the larger number of vertices which must be handled. For one of the test examples we used that is taken from real life, students from all grades are not expected to walk more than 0.5 km to the bus stop [10]. In addition there is an interpretation that routes should not be extended to accommodate particular students [11]. However it appears that in practice often each student is granted his or her own stop since many stops have only one or two students. This is partly a result of the parents of the students threatening to direct their taxes to another school board if their individual requests are not met.

The algorithm used to find the shortest path tree is based on Dijkstra's algorithm given in [1]. V is the set of vertices or nodes and R is the set of vertices still to be considered by the algorithm. P is an array where the index is a node and the value is the parent of that node. There is one entry in the array for each node in V . T is also an array where the index is a node and the value is the edge cost from that

node to the root of the tree. The edge costs given by a two dimensional array C are travel times on existing edges, and a large value where there is not an edge.

Algorithm for `shortestPathTree`

```

 $n := |V|$ 
if unique network node with 0 demand
    then  $root :=$  node with 0 demand
    else  $root :=$  item selected from menu of nodes with 0 demand
 $R := V \setminus \{root\}$ 
for each  $node$  do
     $T[node] := C[node, root]$ 
     $P[node] := root$ 
 $(n - 1)$  times do
     $v :=$  vertex  $x$  in  $R$  such that  $T[x]$  is a minimum
     $R := R \setminus \{v\}$ 
    add  $v$  to tree with parent  $P[v]$ , cost  $T[v]$ 
    for each  $w \in R$  do
        if  $(T[v] + C[v, w]) < T[w]$ 
            then  $T[w] := T[v] + C[v, w]$ 
                 $P[w] := v$ 

```

Using the shortest path tree, the network can be separated into inner and outer districts. Separating the network into inner and outer districts allows us to use a bus for a second time. If a bus completes a route starting in the outer district and drops the students at the school, then in the time that the students are allowed to wait at the school, the bus could complete a closed route from the school fully within the inner district. Those nodes which can be reached and the students transported to the school in this time define the inner district and the other nodes are in the outer district. The algorithm concentrates on the outer district because it is assumed that there are more students further from the school than nearer the school.

2.2.2 Lower and Upper Bounds

Before we start generating routes and timetables we can quickly determine a lower bound and an upper bound on the number of buses needed. A lower bound is found by assuming that all the students waiting in the outer district are grouped together at one imaginary point. This point is at the inner edge of the outer district so that the students are still in the outer district but their travel times are minimized. It is assumed that there is an edge from this point to the school. Then the buses are filled starting with the largest until all the students are on board a bus. The number of buses used gives us the desired lower bound. The number of students in the inner district is also calculated to check that there are more students in the outer district.

There is an additional check if students must be dropped off when they first reach the school. If it is the case that students must be dropped off on reaching the school the network can be split into *isolated components*. Isolated components are maximal components of the network where any two vertices in the component are connected by a path not including the school, as illustrated in figure 2.1. Thus two vertices not in the same isolated component are connected by a path which must include the school as one of its vertices. When students must be dropped off on first reaching the school, the desired minimum number of buses is the sum of the minimum number of buses needed for each component.

An upper bound to the number of buses needed is found by forming naïve routes using the nodes and edges in the shortest path tree. Start with the furthest node in the tree which does not have all its students accounted for, form a route starting at this node with the largest remaining bus from the bus fleet and pick up students on the shortest path from the starting node to the school. All the students at each stop are picked up until the last stop before the school where only some of the students may be taken on board to fill the bus to its capacity. For the purposes of finding the upper bound, whether or not nodes can be split is ignored. The network is updated by removing the students just routed from their stops and the bus is removed from the fleet. This process is repeated until all students are included in a route or the bus fleet is exhausted. If the bus fleet is exhausted then if all the outer-district

Figure 2.1: Isolated Components of the Network

students have been routed, the entire bus fleet is sufficient, but some buses need to perform some closed inner routes. Otherwise the number of buses is insufficient for this naïve algorithm, though it may still be sufficient for a more sophisticated algorithm. If the number of buses is sufficient then timetables are formed describing which bus picks up how many students at which stop at what time.

2.2.3 All Pairs Shortest Paths

A preliminary step used when the routes are found is to find the distance and path between each pair of vertices in the network. This is the all pairs shortest paths problem. There are two main approaches to the problem. One is to find the shortest path tree for each vertex in the network. The other is to use Floyd's algorithm which finds the cost of going from vertex i to vertex j via vertex k where k takes on the value of each vertex in turn. The execution time of the multiple-Dijkstra algorithm is $O(n^3)$ for the simplest implementation for sparse graphs. This could be improved to $O(nm \log n)$ if the algorithm were implemented using an adjacency list to represent the network and if the set of remaining vertices were represented as a priority queue implemented as a partially ordered tree. This compares to $O(n^3)$

for Floyd's algorithm. However Floyd's algorithm has a low constant factor. The main $O(n^3)$ step in Floyd's algorithm is also relatively simple which has advantages for the implementation. Thus Floyd's algorithm was chosen.

The all pairs shortest paths algorithm is based on the modified version of Floyd's algorithm given in [1]. The first step is to number the vertices and then use these index numbers instead of the actual vertices in the cost array. This makes for a quicker implementation and a saving of space. n is the number of vertices in the network, C is an $n \times n$ matrix of edge costs, P is an $n \times n$ matrix of vertex indices. The vertex index at $P[i, j]$ represents the 'mid-point' vertex, or intermediate vertex, on the path from i to j . The shortest path from i to j can be reconstructed by recursively finding the mid-points of the path. The edge costs in the network are given by the matrix C_{Net} . As in the shortest path tree algorithm, the cost on an edge is the travel time for the edge.

Algorithm for allPairsShortestPaths

```

 $n := |V|$ 
for  $i := 1$  to  $n$  do
  for  $j := 1$  to  $n$  do
     $C[i, j] := C_{Net}[i, j]$ 
     $P[i, j] := nil$ 
for  $k := 1$  to  $n$  do
  for  $i := 1$  to  $n$  do
     $costIK := C[i, k]$ 
    for  $j := 1$  to  $n$  do
      if  $(costIK + C[k, j]) < C[i, j]$ 
        then  $C[i, j] := costIK + C[k, j]$ 
            $P[i, j] := k$ 

```

Having found the shortest path tree with the school as the root, and found all pairs shortest paths, this means that the information has been found twice for the shortest path from each node to the school. However to construct the shortest

path tree from the all pairs shortest paths matrices would probably take as long as using Dijkstra's algorithm directly. The shortest path tree is a crucial component of the algorithm so this must be found in either case. The shortest path tree is not immediately obtainable from matrices C and P because the shortest path tree needs the parent of each node on the shortest path to the route, but matrix P only gives some 'mid-point' node. Thus if matrix P were used, it would take a number of steps to find the parent of a node in the shortest path tree. This would have to be repeated for every node. Even a more efficient algorithm which found complete paths to the route so that a number of node-parent pairs could be found at once would only be similar to improvements to Dijkstra's algorithm. Conversely, it would not save much work to use the shortest path tree in the all pairs shortest paths algorithm. The initialization steps could use the shortest path tree since the costs can be found from the tree and the 'mid-point' node is the highest numbered node between the two end points of the path. However, in the main $O(n^3)$ step each of the loops would only be reduced by one step, so that the execution time is still $O(n^3)$.

2.2.4 Route Generation

With the preprocessing steps completed, we can now proceed to finding the routes. The routes are generated in a sequential manner from scratch. The routes are found so as to satisfy all the constraints, rather than finding a route and testing its compatibility with the constraints. The schedules can be generated simultaneously as the routes are found. In other words a timetable is generated for each route. Each timetable is generated individually and completely. There is no backtracking if it later appears that a previously generated timetable increases the costs for later ones. The only adjustment that is made to an outer route is to have a bus arrive earlier at its stops than the timetable initially generated, if the same bus is used for an inner route. The top level algorithm for finding the outer district routes is as follows.

Algorithm for optimumBuses

```
while there are students in outer district do
    find furthest remaining branching point
    if branching point is isolated fork
        then solve corresponding knapsack problem
        else use direct route with detour
```

Isolated forks and the knapsack problem are described below. The check for students in the outer district is made simply by searching the nodes in the outer district which still have students which are not part of a route.

2.2.5 Furthest Branching Point

The branching points from the shortest path tree are found once. They are collected and sorted by their distance to the root, starting with the furthest. The network is also split into isolated components as discussed in section 2.2.2. Any branching points that are in isolated components (apart from the largest one) are given priority over those in the largest component. The branching points in isolated components are also sorted by their distance to the root. The isolated components may be much smaller than the largest component so there is more chance of routing all the students in the component if the largest buses in the fleet are available. They are not considered totally separately since depending on whether students must be dropped off on first reaching the school, there could be routes which include nodes from more than one component. We can also define isolated components of the outer district. These isolated components are maximal components of the network where a path between any two vertices only includes vertices from the outer district. It may be possible to find routes which include nodes from more than one isolated component of the network by going via inner nodes. However considering isolated components of the whole network may help to break the problem down into smaller problems.

The branching points are used in turn, starting with those in the small-size isolated components, as the starting point of a new route. The same branching

point is used until all the students in the descendants of the branching point in the shortest path tree have been included in a route. (Details of the algorithm appear on page 96 in appendix A.)

2.2.6 Isolated Fork

How the route is generated from the branching point depends on whether the *fork*, or subtree of the shortest path tree rooted at the branching point, is isolated in the network (see figure 2.2). If it is isolated then there are no edges in the original network between the nodes of the fork, other than the branching point, and nodes outside the fork. Also there are no edges between nodes in different branches of the fork. We find this out by checking all the descendants of the branching point and checking that leaf nodes in the shortest path tree have degree one in the network and that inner nodes have degree two in the network. If there is another branching point in the fork it is checked that there are no edges between the branching point and nodes outside the fork. If the branching point is the root of an isolated fork, then the nodes of the subtree define a knapsack problem.

Figure 2.2: Isolated Forks

2.2.7 Knapsack Problem

Since the fork defined by the branching point is isolated we can select the starting node of a route by finding the furthest node of the fork which still has students which are not included in a route. If there is more than one node which has the

greatest distance to the root of the fork, then we take the node with the lowest degree in the network. Thus preference is given to nodes which are leaves in the network. Starting at this furthest node proceed directly to the branching point and then proceed to the school. If the largest remaining bus in the bus fleet can be filled with students from the route between the starting point and the branching point then the timetable is formed.

Otherwise the limit of a 0-1 knapsack problem is defined by the maximum allowed riding time minus the time to get from the starting point to the school, divided by two. The objects to 'fit' in the knapsack are the travel times from nodes in the fork which are currently leaves to the branching point. The current leaves are those nodes in the fork which have students waiting but all its descendants in the fork do not have students waiting. In other words these travel times are the times for a bus to travel down a branch of the fork. We want to fit as many branches as possible in the route within the remaining time. With the solution to the knapsack problem we can use up as much of the remaining time as possible to pick up more students. Since the fork is isolated, for each current leaf in the knapsack solution, the route formed must have the bus start at the branching point, proceed to the current leaf and return to the branching point. We do not have to be concerned with other branching points in the fork since these branching points are more distant from the root, so the nodes in their forks will have been included in previously generated routes.

The knapsack problem is solved exactly. The heuristic algorithm due to Ibarra and Kim [24] was investigated but it was not used. It is a heuristic algorithm so it does not always give solutions that are very close to the optimum and it tends to give only one reasonable solution. In this problem there are additional constraints which must be satisfied so it is useful to have additional optimal solutions if one does not satisfy the additional constraints. In addition the heuristic algorithm may not be suitable where the weights for the objective function and the constraints are equal for each possible value in the solution. Thus exact solutions to the knapsack problem are found by enumerating all the permutations. This is an exponential

algorithm but it does give exact solutions and it is assumed that in reality any fork will be small.

Then the solutions generated for the knapsack problem must be tested for their feasibility with respect to bus capacity. The solutions are tried in turn until one is found where the number of students on the route from the starting point to the branching point plus the number of students in the knapsack solution, can be accommodated on an unused bus from the fleet. In addition it may be possible to pick up more students in the outer district on the direct route from the branching point to the school. The smallest bus which can transport all these students is chosen from the fleet. Now the timetable can be made from the route and the bus.

The timetable for an isolated fork is generated in the same manner as for routes which can fill the bus with just a direct route to the school and those which require a solution to the knapsack problem. First the current leaves of the knapsack solution are sorted by the number of students at each leaf node who are not part of another timetable, with the nodes with least students first. This is so that where there are more students at a leaf, the bus goes to them later and fewer students have to wait on the bus while other students are picked up.

So once the route has been formed we can now complete the scheduling part of the algorithm by forming the timetable. The stops added to the timetable do not necessarily include picking up students at that point. Nodes where no students are picked up are used to indicate the complete path through the network and the time that the bus is at each node. The timetable initially starts at the leaf chosen as the starting point of the route. Once the bus reaches the branching point, the next node considered is the next leaf determined by the knapsack solution. The bus actually makes a trip in both directions along the branch but when the timetable is formed the branch is traversed only once. The initial starting time of a route is the starting time of classes minus the time to complete the direct part of the route and the time to visit the leaves in the knapsack solution. Students are picked up along the way until either the bus is full or there are not enough empty seats for all the students at a node, depending on whether nodes are splittable or not. At

each node the current stop is added to the timetable, that is the current time with the number of students boarding the bus at the current node. Also when visiting a node on a path from a leaf of the knapsack solution to the branching point, a stop is added to the timetable to indicate the time that the bus passes the stop on the way out from the branching point to the leaf. No students are picked up on the way out. The next node considered is the parent of the current node in the shortest path tree. The new time becomes the current time plus the time from the node to the parent. When the bus reaches the branching point the next leaf is selected and the time incremented by the time to reach the leaf.

On reaching the branching point for the final time, students are picked up from the branching point if possible, and from the other nodes on the direct route from the branching point to the school. A stop is added for each node visited on the way to the school.

As a final check the number of students picked up by the bus from the outer district is compared with the capacity of the largest bus from the fleet, including the present bus. If the largest bus can be filled then in a greedy sense this is an optimal route since to approach the minimum number of buses needed, we would like to fill the largest remaining bus for each successive timetable. If this is not an optimal timetable then the timetable is marked as such in case the user wishes to modify the timetable generated. (Details of the code for the knapsack problem and the timetable generation appear on page 98 in appendix A.)

2.2.8 Detour routes

If the fork is not isolated, i.e. there are edges between branches in the fork or there are edges from nodes in the fork to nodes outside the fork, then we use routes with detours. As for the isolated fork case the starting point of the route is the descendant of the branching point which is the furthest from the school and still has students not in a timetable. If the number of students in the outer district on a direct route along edges of the shortest path tree to the school can fill the largest remaining bus

in the fleet, then the timetable is made from the direct route. As with the knapsack timetables, if the largest remaining bus can be filled then this is optimal.

Type 7 Detour

If a direct route cannot fill the largest bus then the objective is to find a route which can pick up all the students in the fork which are at nodes in the outer district, up to a maximum of the capacity of the largest remaining bus. The direct route is lengthened by finding a *type 7* detour. This type of detour is so-called because a typical route of this type will have a 'straight' section for the direct part leading to the school and a separate detour leg, forming a figure 7, as shown in figure 2.3.

We can define a circle of nodes which can be reached from the starting point of the direct route. The centre of the circle is the starting point of the direct route. The radius is the time difference between the maximum allowed riding time and the time to travel the direct route. Using the costs from the all pairs shortest paths routine, the nodes within this circle can be found. The nodes should also be in the outer district and there should also be students still waiting to be picked up. Also the nodes should not be part of the direct route. Those nodes remaining in the circle are potential starting points of a detour.

Each of the potential starting points and its shortest path to the direct-route starting point is investigated for the number of students which can be picked up. The maximum number that can be picked up is the capacity of the largest remaining bus minus the number of students in the outer district of the direct route. Detour routes that pass through the school when these are disallowed are removed. Otherwise the number of students at each node in the outer district of the detour route is summed. The route which gives the maximum number of students up to the bus capacity limit is found, with preference given to routes which do not have a split node if these are allowed.

The combined detour route and direct route is then shortened if necessary. This can be necessary if there are nodes in common between the two sections, since travelling from the starting node of the detour to the starting node of the direct

route may entail travelling along part of the direct route in reverse. By splitting the routes into pieces containing the common nodes and those without common nodes, the segments can be put together with the lowest cost. Thus the starting point of the final route could be the starting point of the direct route with part of the detour section being traversed twice; or the starting point could be the starting point of the detour route with part of the direct route being traversed twice.

Zigzag Detour

If the largest remaining bus in the fleet can be filled with students from the outer district of the direct route plus the type 7 detour, then the timetable is made. Otherwise an additional detour is considered. This is a *zigzag* detour, so-called because the route typically will divert to a node away from the current direct route plus type 7 detour and then return to this route using the same path, forming a zigzag, as shown in figure 2.3.

Again the limits to the detour are provided by the remaining seats to be filled on the bus, and the time difference between the maximum riding time and time to traverse the partially formed present route. This time difference is halved to allow for a detour to another node and the return to the route found so far. The nodes which are considered for inclusion in the route are those not already on the route which are in the outer district with students waiting and can be reached in the remaining time. Also the number of students waiting must be compared to the remaining capacity of the largest bus, and the path to a prospective node checked if detours should not pass through the school.

Those nodes which could still be visited on a detour are checked starting with the node with the largest number of waiting students. The current type 7 detour route is checked to find which node should be the start of the zigzag. This node should be the one to minimize the cost of the zigzag. The zigzag is inserted in the route with the shortest path between the two nodes. This process is repeated until there are no more nodes to detour to in the remaining time or the bus is filled. The timetable can now be made with the completed zigzag route.

Figure 2.3: Direct and Detour Routes

Timetables

The timetable is constructed from the route in a straightforward way: use the smallest bus necessary, start at the starting of classes minus the time to travel the route at the start of the route. Each node on the route is visited in order, the time is incremented by the time to traverse each edge and the students picked up if it is the last visit to the node in the outer district. If there are still empty seats after the last node in the outer district was served, then students at nodes on the route which are in the inner district can be picked up. If the largest bus in the fleet was used in the timetable and there are no empty seats at the end of the route then the route is optimal in a greedy sense. Otherwise the non-optimal route is noted as such. (Details of the detour route generation start on page 101 in appendix A.)

2.2.9 Inner District Routes

Once all the students in the outer district are part of a timetable, those students remaining in the inner district can be allocated to timetables. The number of buses should not need to be increased since by definition the students in the inner district should be part of timetables which reuse buses from the outer district. Since the inner routes start at the school and end at the school, we can find the routes by

solving several closed traveling salesman problems. We do this by forming a route to the furthest node in the inner district which still has students, and then using an insertion heuristic [36] to extend the route within the limits set by the travel time and the capacity of the bus. The selection criterion for the next node to insert in a route is to select from those nodes which are reachable in the remaining time, the one with the largest number of students. This does not give optimal traveling salesman tours but the inner routes are usually short. Then the timetable is constructed in a similar manner to the other timetables. This is repeated until all the students are incorporated in a timetable. An adjustment must be made for those outer district timetables where buses are reused since they must now reach the school in time to perform the inner district timetable.

2.3 Measures of Quality

There are three ways in which the quality of the solution is measured. The first of these is to find the difference between the total capacity of the buses used in the solution found by the algorithm above and the total capacity of the buses used for the lower bound. The other two measures measure the inconvenience to the students. In an ideal solution each student would be able to go directly from his or her stop to the school, as in the routes found for the upper bound. However in the solutions generated by the algorithm described above there will almost inevitably be some detours. The *arrival inconvenience* for each student is the difference in time between the time that the student is picked up in the algorithmic solution minus the time that the student would be picked up if the route were direct. The total arrival inconvenience is the sum over all students in each timetable of each arrival inconvenience. The *waiting inconvenience* is similar only it measures the extra length of time that the student has to spend between leaving home and classes starting, above the time spent on a bus on a direct route arriving at the time that classes start. In other words the time spent on detours plus the time spent waiting at school before classes start. In addition to these measures of quality, there are those

timetables which were already marked as being non-optimal which can be examined.

2.4 Algorithm in Summary

Algorithm for rural school bus routing and scheduling

find shortest path tree by Dijkstra's algorithm to also find:

inner and outer districts, leaves of the tree

find isolated components of the network: grow node sets by depth first search

sort branching points of shortest path tree by distance

find all pairs shortest paths by Floyd's algorithm

while there are students in the outer district do

 get furthest remaining branching point ($node_b$)

 if $node_b$ defines isolated fork then

 get furthest remaining node ($node_f$) in fork

 if demand from $node_f$ to $node_b$ fills largest bus then

 make timetable

 else

 find limit to knapsack problem:

$(\text{maximum riding time} - \text{time from } node_f \text{ to school}) \div 2$

 knapsack elements are times to current leaves on isolated branches:

 solve knapsack problem exactly, sort solutions by travel times

 test knapsack solutions in order until demand on $node_f$ branch

 + demand from knapsack solution nodes fills a bus

 get smallest size bus necessary, fill empty seats from $node_b$ to school

 make timetable: visit leaf nodes starting with most distant

```

else (not isolated fork) if direct route can fill bus then
    make timetable
else use type 7 or zigzag detour to lengthen direct route
    get furthest remaining node ( $node_f$ ) in fork
    find nodes reachable in remaining time (max time — direct-route time)
        from  $node_f$  in outer district and with students remaining
    for each  $node$  in reachable nodes do
        find number of students in outer district on path from  $node$  to  $node_f$ 
        remove detours including school if these are not allowed
        select detour path with maximum students up to remaining bus capacity
        rearrange combined detour and direct route segments if necessary
        if maximum capacity bus can be filled then
            make timetable of direct and type 7 detour
        else use zigzag additional detour
            find nodes in outer district reachable in remaining time from nodes on
                route which can fill remaining bus capacity and where route does not
                pass through the school if disallowed
            find reachable node with maximum number of students and
                start of detour minimizing detour length
    endif — end of route generation
as each timetable is made check for split nodes if allowed to fill bus
endwhile — all outer district students routed
for students remaining in inner district solve closed traveling salesman
    problems with insertion heuristic reusing buses from outer district routes
for each timetable with bus making second trip
    find time for bus ( $bus_c$ ) to make closed route ( $time_c$ )
    subtract  $time_c$  from stop times of outer district timetable using  $bus_c$ 

```


Chapter 3

Implementation and User Interface

The algorithms described in chapter 2 have been implemented in the Smalltalk language. This language is chosen because it provides a good programming environment particularly including an implementation of windows and basic graphics and it provides mouse support. Smalltalk also provides a convenient way of organizing the code by requiring the user to separate the code into methods appropriate for particular classes. The language is slower compared to C, for example, for implementing algorithms but Smalltalk also allows the user to write user-defined primitives written in another language where speed is important. This has been done for the all pairs shortest paths algorithm where the $O(n^3)$ part of the algorithm has been implemented in C.

3.1 Data Input

The data for a particular school bus routing and scheduling problem is input in a Smalltalk-like syntax. This has the advantage of being more intelligible to the user than specifying a series of parameters in a predefined order and yet still being 'computer readable'. For small problems the user can also enter the data directly by adding nodes and edges to a new network (see section 3.3).

The example input partially shown here is for a school district around Gerolstein in West Germany. For this problem the nodes are relatively far apart and students are generally required to assemble at a central bus stop in the village or hamlet.

For this example the data was entered by creating an input data file. The input starts by creating a new global variable which contains the network, the buses, the parameters for the particular problem, the display and the timetables when they are generated. Here `BusRoute` signifies the whole set of data and not just an individual route used in a timetable. This is done in Smalltalk by assigning `GerolRoute` to be a new initialized instance of the `BusRoute` class. In other words, `GerolRoute` is the global variable which will contain all the data entered for the problem and generated by the algorithm.

```
GerolRoute := BusRoute new.
```

Then the nodes in the network are added. Each new node is given a name, a position found by finding the coordinates of the node on a real life map and the demand at the node. The map coordinates are simply specified as a pair of x and y coordinates, so that the numbers given come from the map of the Gerolstein area. The coordinates play no direct part in the algorithm but are used for the graphical display so that the display fairly accurately resembles a map. However it may be desirable for nodes to be moved on the display so that nodes may be more clearly seen if their map coordinates would place them too close together on the display. Moving them of course has no effect on the algorithms. (The coordinates are given in the form of a string because of a compiler error.) The node is then added to the bus route. Three examples are shown below. The school is at Gerolstein hence there are no students waiting for a bus at that node.

```
GerolRoute addNode:
```

```
  (NetworkNode new name: 'Berlingen'  
    position: '273 @ 827' demand: 46).!
```

```
GerolRoute addNode:
```

```
  (NetworkNode new name: 'Michelbach'
```

```

        position: '236 @ 769' demand: 4).!
GerolRoute addNode:
    (NetworkNode new name: 'Gerolstein'
        position: '235 @ 808' demand: 0).!

```

The nodes are then connected together in the network by specifying the edges between the nodes and the distance (in tenths of a kilometre) along the edge. The edge is specified by naming the end points using the names of the nodes already entered. Also the driving environment or edge set F_i is specified by making the connection either a village connection or a regular, i.e. rural, connection. In the Gerolstein example all edges are in a single rural edge set. Three examples of edges are shown below.

```

GerolRoute connect: 'Berlingen' to: 'Pelm' distance: 35.
GerolRoute connect: 'Gerolstein' to: 'Pelm' distance: 25.
GerolRoute connect: 'Gerolstein' to: 'Roth' distance: 45.

```

Then the parameters are specified. First there are the time restrictions T_{start} , T_{ride} , T_{wait} . Then there are the average bus speeds (at present only rural and village speeds can be entered), $a_{F_{\text{rural}}}$ and $a_{F_{\text{village}}}$. The user can specify whether or not the algorithm should halt at some stages in the generation of the timetables allowing the user to inspect the work so far. Finally the user should specify whether the nodes are splittable or not, and whether or not students must be dropped off at the school when they first reach it. The values for the Gerolstein example are shown below.

```

GerolRoute startingTime: (Time fromString: '8:15:00').
GerolRoute maximalRidingTime: (Time fromString: '0:45:00').
GerolRoute maximalWaitingTime: (Time fromString: '00:15:00').
GerolRoute averageBusSpeed: 350.

```

GerolRoute averageVillageBusSpeed: 200.

GerolRoute haltEnabled: false.

GerolRoute nodesSplittable: true.

GerolRoute mustDropAtSchool: false.

Then the bus fleet is specified by adding buses to the existing bus fleet, which is initially empty. Each bus is created and then the capacity is specified and the bus given a name. Three buses are shown for the Gerolstein example.

GerolRoute busFleet

add: (Bus new capacity: 98; name: 'Bus1');

add: (Bus new capacity: 48; name: 'Bus3');

add: (Bus new capacity: 18; name: 'Bus19').

Once the data has been read in, the user can display the data and get access to menus allowing the algorithm to be run, simply by evaluating:

GerolRoute open

in the case of the Gerolstein data. If the user chooses to save the data by selecting 'saveAs...' from the network menu (see section 3.6), the input data is saved to a file in the same form as that described above.

3.2 Network Display

When the data has been input and the window opened, the window shows four panes (see example in figure 3.1). The main pane shows the network of edges and vertices with the vertices shown simply as junctions in the network. The other panes are used when the routes are displayed. The top two show the current time in digital and analogue form. The bottom pane shows the last few pick-ups and drop-offs. More will be said about these three panes in section 3.5. The example shown is the network for the Gerolstein problem.

Figure 3.1: Network Display

There is a menu associated with the display of the network. This menu appears under the name 'Map' on the menu bar. It allows the user to display the network and some or all of the data associated with the network. It also allows the user to edit the network, to magnify part of the network and to save the data associated with the whole problem. Figure 3.2 shows this menu.

Figure 3.2: Network Display and 'Map' Menu

The user can display more network information by selecting an option from the map menu. Selecting 'distances' shows the distance along an edge between two nodes. The demands, or number of students, at each node can be displayed along with the initial letter of the name of each node can be displayed. The full names of each node can also be displayed. Finally the network can be redrawn to display just the network. The same network as in figure 3.1, is shown with the distances, node demands and node names displayed simultaneously, in figure 3.3.

Figure 3.3: Network with Distances, Node Demands and Node Names

When the map or network is displayed with the node demands, node names and the distance along each edge, the amount of information can be too much for the user to see. This also happens with a network where there are too many edges to be seen clearly when the whole network is displayed. To help to see some more detail, the user can choose to magnify part of the network. The user is prompted to enter the corners of a rectangle which is used to select the nodes to display on the network pane. That part of the network selected is only magnified if the magnified section can be enlarged in both the x and y directions within the pane. The proportions of the pane are determined automatically from the coordinates of the selected nodes, and not determined by the dimensions of the rectangle used to select the nodes. The next two figures show the rectangle to select the part of the network to magnify, and the resulting magnified network.

Figure 3.4: Network with Rectangle for Magnification

(The Hohenfels node on figure 3.5 appears to have two names because both the short form of the name and the full name are displayed at once. The short form of the name is always displayed on the right-hand side of the node, but the full name may be displayed on the left if there is insufficient room to display the name on the right of the node. When the name is displayed on the right-hand side simultaneously with the short form, then the first four letters will overlap so the simultaneous display is not apparent. When it is displayed on the left-hand side then the last letter or two of the name overlaps with the demand number. The placement of the nodes on the pane is determined automatically so the user would have to select a different section of the network to magnify in order to change the distance of the node from the edge of the pane.)

3.3 Network Editing

The map menu also has options to allow the user to change the network. Nodes and edges can be added or removed. Also the initial number of students at a node can be changed. This is not only so that the user could enter a network using just these facilities, but also so that changes can be made and the problem re-solved. This allows for an existing network to be updated as new students enter the school district and for new edges to be added as new roads are built. It also allows the user to see what timetables are generated as a result of, for example, moving the students from one stop to another.

When the user selects the 'addNode' option, the user is first prompted for the name of the node to add. If this name is one of the nodes already part of the whole data set, e.g. GerolRoute, and not part of the network, then the node is added to the network. It is possible and may be useful for editing purposes, to have nodes which are part of the whole data set but not part of the network. If the node is not already known, then the user is prompted for the position on the real life map of the node and the number of students at the node. A node is then created from this data and it is added to the data for the BusRoute instance and the network. The network display is then updated to show the new node, as it is when other network editing changes are made.

If the user chooses to remove a node then a small rectangle appears on the screen for the user to place around the node to remove (see figure 3.6). If this includes more than one node then a menu appears for the user to pick the correct node to delete. If there are edges from the deleted node to the remainder of the network, then those edges are deleted.

Figure 3.6: Network after User has selected to Remove a Node

When the user wants to add an edge, the user is prompted for the end points of the edge in the same manner as for removing a node. The user also selects the type of edge from a menu and the distance along the edge. Similarly, when the user wants to remove an edge, the user is prompted for the two end points of the edge.

The 'make' option on the 'Execute' menu allows the user to create a route by hand. The user might want to create a route by hand if, for example, the user needs to fix a route before the rest are generated. The user is prompted for the name of the timetable, the bus to use for the route and the starting node for the route. If the first node is the school then the route is closed. The user enters each node for the route in the usual manner for selecting a node and enters the number of students to pick up at each stop if the demands are splittable. The route is finished when

the school is reached. The route is checked to ensure it is not too long. If the route is closed and if there is an existing open route which uses the same bus, then the open route must be adjusted to allow it to be completed before the closed route starts. There is a check to make sure that this does not result in a timetable which starts too early. At present this facility only allows the user to create a timetable, but it could be fairly easily modified to allow the user to create some timetables and then run the algorithms without the students which are already accommodated in the user-created timetables. However the present facility could be used by the user to create a complete set of timetables. This would mean that the user would be able to indicate the desired routes, but the constraint checking would be done automatically. The user could then compare the number of buses needed for the manually created set of timetables to that found heuristically.

3.4 Algorithm Execution

The algorithms discussed in chapter 2 are made available to the user through the menu labelled 'Execute'. This menu is shown in figure 3.7.

Figure 3.7: 'Execute' Menu

First the shortest path tree can be generated and displayed. The shortest path tree is stored so that it can be used in subsequent routines. If the network is magnified then the travel times on each edge are shown. For the example shown in figure 3.8, the whole network is magnified so all the edge costs are shown.

Figure 3.8: Shortest Path Tree with Travel Times

If the user then wants the upper and lower bounds to the number of buses needed to solve the particular problem, these can be seen using the 'min' and 'max' menu options. The answers are displayed in a separate window. The main heuristic algorithm is invoked by selecting the 'best' menu option. Again the number of buses needed is displayed in a separate window along with the execution times of various stages of the algorithm, as shown in figure 3.9.

Figure 3.9: Window with Results of Heuristic Algorithm Execution

A display of all the bus statistics for the route, i.e. the upper and lower bounds and the value found heuristically, can be displayed by selecting 'stats' from the 'Execute' menu (see figure 3.10). The statistics window also shows in rudimentary way the changes that have been made to the network starting from a base of the input data file. This shows the nodes and edges that have been added or removed from the original problem so that the user can see which particular network the statistics relate to.

Figure 3.10: Window with Bus Statistics

This menu also allows the user to see the *defects* in the timetables generated. That is the user can see the difference between the number of buses needed in the lower bound and those needed in the heuristic solution (Δ_{bus}), and the arrival and waiting inconveniences (Δ_{arrive} and Δ_{wait} respectively). The values are again displayed in separate windows, as shown for the arrival inconvenience in figure 3.11.

Figure 3.11: Window with Arrival Inconvenience

There is also an option to indicate the non-optimal routes. These are displayed in a rudimentary way so that the user can find those routes which might be improved by hand.

In the 'Map' menu there is an option to reinitialize the network. This clears the shortest path tree and all pairs shortest paths tree and resets the display. The user need only use this if he or she changes a parameter which changes the parameters for the shortest path tree, namely the maximum waiting time and the average bus speeds.

3.5 Route Display

Timetables are generated by the main heuristic algorithm and also as a by-product of using the naïve algorithm to find an upper bound to the number of buses needed. In order to give the user some feel for what the timetables are like in real life, the timetables can be displayed as an animation of the buses on the network. The top two side panes show in analogue and digital form, the current time for the network animation. The network display shows what road each bus is on at the current time, assuming that the timetable has started for a given bus. The direction of a bus on a road is indicated by an arrow drawn on the edge in the direction that the bus is travelling. There is no visual indication of which bus corresponds to which arrow. If there is more than one bus travelling on a particular road in the same direction, then more than one arrow is drawn on the edge. The animation normally starts with the network display showing all the students waiting at their bus stops by displaying the number of students waiting at each bus stop. A bus is not shown stopped at a bus stop when it stops to pick up or drop off students, but as a bus passes a bus stop, the number of students waiting at the bus stop is updated to reflect the number of students picked up by the bus. The bottom side pane also shows the students being picked up or dropped off by displaying the number of students, the name of the bus stop, the name of the bus and the time of the pick-up or drop-off. The network display is animated because the clock ticks periodically, updates the time displayed on the clocks and updates the network display.

The user normally starts the display of the timetables by selecting 'restart' from the 'Clock' menu. This clears the network display to show the basic network and the numbers of students at each node. Before the user starts the animation, the menu option to just turn the clock on is disabled. The 'Clock' menu is shown in figure 3.12.

Figure 3.12: 'Clock' Menu

The screenshot shows a vintage computer interface. At the top, a menu bar includes options like 'File', 'Edit', 'Smalltalk', 'Execute', 'Map', 'Clock', and 'Window'. The title bar reads 'Gernstein'. The central workspace contains a network graph with various nodes and connecting edges. To the right, a sidebar features a digital time display at '08 02 30', an analog clock, and a scrollable log of events.

The user can alter two parameters affecting the progress of the animation. The 'speed' option allows the user to adjust the amount by which the screen clock is incremented on each clock tick. This allows the user to view the movement of buses in more or less snapshots. The 'tick rate' option changes the number of milliseconds that elapse between screen updates. Changing the tick rate alters the number of milliseconds that a timer process sleeps between sending ticks to the clock. The user can pause the animation by selecting 'clock off' from the 'Clock' menu once the animation is going, and then selecting 'clock on' to start the animation again. The user can also pause the animation by moving the cursor to the text pane displaying the bus movements. This allows the user to scroll over the information. The animation resumes when the cursor is moved out of this text pane.

If the clock is off, the user can also change the time of the animation either by editing the digital time or moving the hands on the analogue face by clicking the

mouse. Changing the time on one pane results in the time being displayed on the other pane changing and the 'clock on' menu option is disabled. This forces the user to replay the animation in one step from the earliest time in a timetable up to the new screen time, thus ensuring that the animation stays in a well defined form.

The animation ends when all the buses arrive for the final time at the school, as shown in figure 3.14.

Figure 3.14: Network Animation with all Students at the School

There is also an option on the 'Clock' menu which toggles the terminal bell on and off. If the bell is on then there is a beep each time that some students are picked up or dropped off, in order to give an extra indication of some activity on the network.

3.6 Data Output

There are two ways in which the data can be output in a textual form. From the 'save' option from the 'Execute' menu. This displays on a window the timetables sorted by the time that the timetable starts, i.e. the time of the first bus stop. The name of the timetable and the name and capacity of the bus are shown, followed by a list of the bus stops showing the name of the bus stop and the number of passengers being picked up or being dropped off. If a timetable has a closed route then the bus is one that has been reused on a inner route. This kind of bus is indicated by it being called an *IBus*. This textual data can be saved to disk by saving the window. The timetables generated for the Gerolstein district are shown in figure 3.15.

Timetable	Bus	Capacity
Timetable2	Bus1	58
Weissenstein	07:36:26	3
Mürlenbach	07:45:00	56
Birresborn	07:51:51	39
Lissingen	08:01:17	0
Gerolstein	08:03:00	0
Timetable1	Bus2	98
Michelbach	07:39:51	4
Büschel	07:46:42	0
Niederbach	07:50:08	3
Büschel	07:53:34	45
Gerolstein	08:02:08	0
Timetable5	Bus4	48
Henert	07:39:52	4
Mürlenbach	07:43:18	0
Birresborn	07:50:09	0
Kopp	07:57:00	15
Birresborn	08:03:51	29
Lissingen	08:13:17	0
Gerolstein	08:15:00	0
Timetable13	Bus15	27
Duppech	07:45:51	9
Scheuern	07:49:17	0

Figure 3.15: Generated Timetables

The second way of saving the timetables is to select 'saveAs...' from the 'Map' menu. This saves to a file all the data for the entire bus route data set in the same form as in the input file. In addition the user can choose whether or not to save the generated timetables as well. They are saved in a form so that the data can be read

in as well as the other input data. Each timetable has the form of that shown below. A new timetable is created and added to the data set, with a bus from the bus fleet and a name for the timetable. Each bus stop is specified by giving the time, the name of the node and the number of students to be picked up at each stop.

```
GerolRoute addTimetable: (  
  Timetable new bus: (GerolRoute busFleet busNamed: 'Bus5');  
  name: 'Timetable1';  
  addTime: (Time fromString: '07:48:26')  
  place: (GerolRoute node: 'Weissenseifen')  
  students: 3;  
  addTime: (Time fromString: '07:57:00')  
  place: (GerolRoute node: 'Murlenbach')  
  students: 45;  
  addTime: (Time fromString: '08:03:51')  
  place: (GerolRoute node: 'Birresborn')  
  students: 0;  
  addTime: (Time fromString: '08:13:17')  
  place: (GerolRoute node: 'Lissingen')  
  students: 0;  
  addTime: (Time fromString: '08:15:00')  
  place: (GerolRoute node: 'Gerolstein')  
  students: 0)..!
```

If the timetable is one for a closed inner route as discussed in subsection 2.2.9, then the bus for the timetable is specified as follows.

```
Timetable new bus: (InnerBus bus:  
  (GerolRoute busFleet busNamed: 'Bus9'));
```

Chapter 4

Computational Experience

4.1 Gerolstein Problem

The program has been tested on three sets of data. In keeping with trying to use as many real-life constraints as possible, the data should be from real life. This applies to the first two sets of data which are based on the same problem. The third set is from a local school board. No other published sets of data were available, particularly data for a rural area. Contact was made with another group working on a school bus problem, but no data was made available. Some random problems could have been generated, but then some care would have had to have been taken to ensure that a 'random' problem was also 'realistic'.

The first data set is for the Gerolstein problem. The problem has been introduced in section 3.1. The complete input data file appears in appendix B on page 107. The data come from a Fortran computer program by Fiala and Lindenberg, supplemented by distances measured from a map of the roads around Gerolstein. There are 27 nodes and 42 edges specified (specified as 84 arcs). The demands at the nodes are often large compared to the number of students that can ride on a bus. Classes at the school start at 8:15 a.m., students are allowed to ride on the bus for a maximum of 45 minutes and they can wait for a maximum of 15 minutes at school before classes start. There is only one edge set — the rural set — and the average speed of the bus on the roads is 35 kilometres per hour. The restrictions on the route construction are at their loosest with the demand at a node allowed to be split among different

buses, and routes being allowed to have the school as an intermediate node as well as the final node. There are buses of many different capacities available from 98 to 18 passengers, though most have a capacity of 48 or less.

The timetables generated by the algorithm are shown in appendix B on page 111. The maximum number of buses needed is 19 and the minimum is 14. The algorithm generates 14 timetables with open routes and 4 with closed routes, so 14 buses are needed. This means that here the minimum number of buses can be reached. The algorithm takes 17.1 seconds to run on a Macintosh IIcx. Of this 5.6 seconds are to find the shortest path tree, and 4.6 seconds to find the all pairs shortest paths matrices, which is 60% of the total.

The timetables generated can also be seen to be close to the optimal value since the bus capacity defect is -9. That is, the difference between the total capacity of the minimum number of buses needed to solve the problem, and the number needed for these timetables. The value can be negative since the buses for the minimum number of buses are chosen strictly in decreasing order of capacity, so the last bus may have empty seats. Also the only non-optimal route, in the sense of always using and filling the largest remaining bus in the fleet, is the last one. The waiting inconvenience and the arrival inconvenience are both large. There are 9 timetables which have a waiting inconvenience, though 40% of the total inconvenience comes from Timetable2 since so many students have a long detour passing through the school before finally being dropped off. There are 10 timetables which have an arrival inconvenience.

When the maximal riding time is reduced to 30 minutes and the average bus speed is reduced to 32 kilometres per hour, 21 timetables are needed of which 16 have open routes and 5 have closed routes, so 16 buses are needed. So two more open routes are needed in this case. The maximum number of buses needed is still 19 and the minimum is 14. Two timetables have considerable waiting inconvenience and a total of 6 have some waiting inconvenience. Ten timetables have some arrival waiting inconvenience.

If the average speed is increased to 38 kilometres per hour then 14 open routes

and 5 closed routes are needed. However this is no longer the minimum since increasing the average speed effectively increases the radius of the inner district so that there are less students to be picked up on open routes. For 38 kilometres per hour the minimum number of buses is 12. Keeping the speed at 38 kilometres per hour and increasing the maximum riding time to 45 minutes results in 13 open routes and 6 closed routes. Increasing the time again to 60 minutes results in 12 open routes and 5 closed routes, so that the minimum can again be attained. Keeping the time allowed to 60 minutes, if the average speed is reduced to 35 kilometres, then 14 buses are needed but the minimum also increases to 14 buses.

The demands at some nodes are high compared to the capacity of the buses so not surprisingly the problem cannot be solved if the demands at nodes are not splittable. Returning to having an average bus speed of 32 kilometres per hour and 30 minutes maximum riding time, if buses must now drop students on first reaching the school, then the minimum number of buses is still 14. The minimum number of buses found in the algorithm is 16. If the maximum riding time is increased to 45 minutes then only 15 buses are needed.

4.2 Lindenberg and Fiala

The second set of data is that from the work of Lindenberg and Fiala [30] which is similar to the Gerolstein data. There are 26 nodes and 35 edges. The demands on the nodes are different from the Gerolstein problem but the values are again large compared to the capacity of the buses. The maximum riding time on the bus is 30 minutes and the maximum time for waiting before classes is 15 minutes. Again there is only one edge set. The average bus speed is 32 kilometres per hour. The demand on the nodes is splittable and routes can have the school as an intermediate node. The bus fleet has 22 buses with a wide range of capacities in gradually decreasing sizes from 95 to 8.

The maximum number of buses is still 19 though the minimum is now 7. The minimum number of buses found by the algorithm is 9, for 9 buses on open routes

and an additional 7 closed routes. In working the data by hand in [30], the authors find that the actual minimum number of buses needed is 9, with 9 open routes and 6 closed routes. There are four timetables where students are inconvenienced with respect to their arrival time at the school, though most is on one timetable. The waiting inconvenience is naturally much larger since there are 7 closed routes, with 8 routes having some inconvenience.

The number of buses can be reduced if the maximum riding time or the average bus speed is increased. With a riding time of 35 minutes only 8 buses are needed, and for a riding time of 40 minutes the minimum number of 7 buses is attained. With an average speed of 35 kilometres per hour 9 buses are still needed. This decreases to 8 buses with a speed of 38 kilometres per hour, with 7 buses reused for inner routes. With a speed of 40 kilometres per hour the inner district has increased to such an extent that the algorithm in the present implementation fails to complete since there are too few buses used for open routes to accommodate those students remaining for inner district routes. When students must be dropped on first reaching the school, the number of buses does not immediately rise. However the maximum riding time has to be increased to 45 minutes before only 8 buses are needed.

4.3 St. Mark High School

The third set of data comes from the actual data used for busing students to St. Mark High School in the Carleton Roman Catholic School Board, located in Manotick, Ontario. The school board supplied timetables and maps showing the routes used for each route. From these maps it is possible to find the positions of the stops where students are picked up and to find the distance between neighbouring stops. Some additional nodes added to the network allow for incorporation of road junctions where there are not any students to be picked up. The distances were measured by plotting the nodes on a map and measuring the distance on the map between the nodes. This naturally gives rise to inaccuracies in the model due to inaccuracies in the original data, such as imprecise descriptions of where a bus stop is, leading

to difficulties in finding the correct position to plot the nodes. Inaccuracies in the distances also arise due to needing to use more than one map to plot the nodes especially for the nodes in the villages in the school district.

There are two edge sets in the data since there are a number of villages in the neighbourhood of Manotick. In the Gerolstein problem it is assumed that students will congregate at one stop in each village where the bus will stop to pick them up. In the St. Mark's case students are much more likely to be picked up at their home. This has two consequences. The first is that buses must make trips within the village itself to pick up students hence the need for two edge sets — rural and village — since the bus travels at a slower average speed within the village. The second is that the demands at each node are much lower than in the Gerolstein problem.

In the model there are 365 nodes including 22 which are used as junctions. There are 515 edges of which 163 are village edges and 352 are rural edges. As already mentioned the demands are lower than in the Gerolstein problem. Most nodes have only one or two students at them and the largest has only 22. Classes start at 8:45 a.m. Students in grades 7 and 8 are allowed to ride on the bus for 60 minutes according to school board policy ([10]). Students in grades 9 through OAC are allowed to ride for 90 minutes. For the purposes of solving the problem a time of 90 minutes has been assumed. There is no specific time given for the maximum waiting time so a time of 15 minutes has been assumed.

For the average bus speeds, the school board timetables were examined to find the average speed for each bus for the given timetable. Assuming that the information given in these timetables is correct, there is a wide range of actual average bus speeds from a low of 28.4 kilometres per hour to a high of 84 kilometres per hour. However most speeds lie in a range of 36 kilometres per hour to 53 kilometres per hour. Estimating average speeds on the low side leads to an average rural bus speed of 40 kilometres per hour and an average village bus speed of 20 kilometres per hour. To start with it is assumed that the demand at a node is not splittable since most demands are small and it may lead to confusion in real life if some students at a

stop get on one bus and some get on another. It is also assumed that students must be dropped off on first reaching the school.

The bus fleet is much more uniform than in the Gerolstein problem. There are 22 buses with a capacity of 72 students, one with a capacity of 66, one with a capacity of 48 and 6 with a capacity of 20. This is because most of the buses are the standard large school buses and vans are the capacity-twenty buses.

The results for the St. Mark's problem are still preliminary but indicate at least that the problem can be solved. With the parameters as above but with demands allowed to be split and buses not having to drop students on first reaching the school, all 30 buses are needed to transport the students to the school. There are 30 buses in the available fleet because the real-life timetables for St. Mark's use these 30 buses. Thus the problem is actually feasible with 30 buses. There are only 5 buses reused for inner district routes, though the inner district is rather small since there are only 22 nodes in the inner district. By increasing the average rural bus speed to 50 kilometres per hour and increasing the average village bus speed to 30 kilometres per hour, increasing the maximum time that students can wait at the school to 20 minutes, but with nodes not splittable and not allowing the school as an intermediate node the number of buses needed is reduced to 25 with 14 buses being reused. However the buses still have a high proportion of empty seats compared to the other school bus problems. Also the minimum number of buses needed with these parameters is 10. As expected, this problem also takes much longer to run, taking about four and three quarter minutes for the shortest path tree algorithm, just over 100 minutes for the all pairs shortest paths algorithm, for a total of about 125 minutes.

Chapter 5

Summary

In conclusion, this thesis has presented an algorithm and its implementation which generates timetables for the rural school bus routing and scheduling problem. The problem is described by a network of nodes and the distances between them, as well as the number of students waiting at each node. The user also specifies the coordinates of each node so that the network can be displayed on the screen with the nodes in the same positions as on a map. The user also specifies the time windows within which the students must be picked up. The allowance for students to wait at the school before classes start means that an inner district can be defined so that some students can be picked up when a bus makes a second trip. The edges of the network can be classified as rural or village, thus allowing different average bus speeds on rural or village roads, and for the recalculation of the edge times from their constant distances. The remaining two parameters are whether to allow demands to be split and whether students must be dropped at the school as soon as they get there.

The main algorithm is based on that by Lindenberg and Fiala [30]. Before running the main algorithm the user can find the minimum and maximum number of buses needed to transport the students to the school. This helps to give a context in which to place the result of the heuristic algorithm. The main algorithm then attempts to find the least number of buses needed to get the students to the school within the constraints. If there are any isolated components then these are routed first. Then the routes to the school follow as much as possible the paths on the

shortest path tree so that students get to school as quickly as possible. Where a bus cannot be filled by a direct route then either a route is formed from a solution to a knapsack problem where the most distant branching node defines an isolated fork. or the direct route is lengthened by first using a type 7 detour and then a zigzag detour. Finally any students not already in a timetable with an outer district route are picked up by a bus running a second closed route.

As mentioned earlier (section 2.2), the following features of the Lindenberg-Fiala model and algorithm are also used in the algorithm: (1) the objective is to find the minimum number of buses; (2) T_{ride} , T_{wait} , and T_{start} ; (3) a heterogeneous bus fleet; (4) only one school; (5) inner and outer districts; (6) type 1 detours are similar to type 7 detours; (7) routes are based on the shortest path tree and the tree can be used to find branching points; (8) finding isolated components of the network; (9) finding lower bound to number of buses; (10) using isolated forks to solve a knapsack problem; (11) using closed routes in the inner district. There also a number of differences including: (1) edges are split into subsets each with its own average bus speed rather than an average bus speed for the whole network; (2) whether nodes are splittable is a parameter rather than assuming it is allowed; (3) whether the bus can pass by the school without stopping is a parameter; (4) the shortest path tree uses time as the cost rather than the distance; (5) finding an upper bound to the number of buses; (6) type 7 and zigzag detours are used instead of type 1, type 2 and type 3 detours; (7) instead of k -blocks, extensions to the direct route are tried; (8) no emphasis on keeping within the minimum bus representation. In addition the implementation of the whole algorithm is completely new.

The graphical interface, which is also completely new, allows the user to enter data either with a mouse and the graphical display, or by entering the data with a Smalltalk-like syntax from a file. The network is displayed on the screen and the user can selectively display the names of the nodes, the distances between the nodes and the demand at each node. The user can modify the network by adding or removing both nodes and demands with appropriate prompts for each operation. Also the demands can be changed. The network can also be magnified so that crowded

regions can be seen more clearly. Once the user has generated the timetables, where as few buses are used as possible, there are two measures of inconvenience that can be displayed, as well as the bus defect. Where a timetable does not satisfy some subjective criteria, it could be removed from the solution and a new one created manually on the screen. The timetables can be saved in a formatted way or saved to disk. Finally the timetables can be shown on the network in an animated way so that the buses move, with two clocks to show the current time being shown on the display. This lets the user see how the buses are moving for the whole network or for some part of it.

The algorithm has been tested on three different networks each being tried with different values for the parameters. For the two smaller networks, using reasonable values for the parameters, the actual minimum number of buses can be attained, though the number of buses used by the algorithm is fairly sensitive to the values chosen. The final network was much larger and makes use of two edge sets unlike the first two network. The algorithm does not do so well here but can still find a feasible solution.

Chapter 6

Future Work

There are a number of changes that could be made to the algorithm and the implementation if any work is done in the future. The slowest part of the algorithm is the preprocessing steps to find the shortest path tree and all pairs shortest paths. These steps may not be considered preprocessing if the user wants to keep making changes to the network. In the present implementation when any change is made to the network, the shortest path tree and the all pairs shortest paths matrices are discarded. In general changing one edge of the network could change the shortest path tree but in practice there could be some speed up by constructing a new shortest path tree using as much of the old shortest path tree as possible. The all pairs shortest paths algorithm is the slower of the two and it has already been implemented in the C language to be called as a primitive from Smalltalk. The same might also be done for the shortest path tree. In addition the actual algorithms used might be re-examined to see if another algorithm could be used which runs in less time. This might include not finding all the paths for the all pairs shortest paths step but only find the paths as needed.

A useful review of shortest path algorithms, their data structures and their implementation appears in [16]. Rather than use the basic implementation of Dijkstra's algorithm, the set of nodes to be explored on each iteration could be kept ordered. Possibilities include having the list totally ordered, using a radix sorting technique and using a priority queue implemented using a binary heap. The minimum element is respectively: the first element of the ordered list, the minimum element of

the first non-empty bucket and the top element of the heap. Also lists could be used: both a queue and a deque (a double-end queue). In the deque case the nodes are partitioned into two subsets which are handled differently. In both cases the node examined at each iteration is the node from the head of the list. The worst case complexity of these algorithms is respectively: $O(ne)$, $O(e + nc_{\max})$, $O(e \log n)$, $O(ne)$ and $O(n2^n)$, where $c_{\max} = \max\{c_{ij} : (i, j) \in E\}$. However for real-life cases the worst case complexity may be misleading, such as for the deque case which has an exponential complexity because of a few pathological cases but turns out to be one of the fastest in real-life problems. Graphs for real-life problems are sparse, nearly planar and large. For rural-type networks, sparse random graphs are nearest to real life. The worst performance is by Dijkstra's algorithm itself, using a totally ordered list is somewhat better, and using a heap is better still. The radix sort algorithm is strongly influenced by the range of edge costs and is best when $c_{\max}$ is small. The queue implementation is rather efficient and the running time increases almost linearly with the size of the graph. It is sensitive to the size of the network, particularly the diameter of the graph. The diameter is defined as the maximum of the minimum number of edges connecting any two vertices in the graph. The running time for the deque implementation is as fast as the queue but is less dependent on the diameter of the graph. Thus the deque implementation would be a suitable candidate to test here.

There is also some comment on reoptimization when the shortest path tree is needed for a network which is only slightly different from another network. They conclude that using reoptimization techniques using the slightly different shortest path tree is advantageous when small length changes occur involving just a few arcs. They also present two algorithms for the sequential approach to the all pairs shortest paths problem which can be implemented efficiently, rather than using Floyd's algorithm.

As each route is found at present, the only optimization that is done is to try rearranging the order of the detour and direct segments. This could be improved to completely reform the route by finding the shortest path through all the nodes on

of/de

2

the route which have students to be picked up. Another way of posing the problem is as a traveling salesman problem which has an open tour with one end point fixed. Solving this problem exactly would mean making more use of the all pairs shortest paths matrices, which may have an impact on the strategy used with regard to changing the algorithm for finding these matrices.

At present when the knapsack routes are formed, any spare capacity on the bus is filled by picking up students on the direct route from the branching point to the school. This could be improved by allowing for possible detours between the branching point and the school.

At present only two edge sets called rural and village are permitted. This should be improved first of all to allow for a fixed larger number of edge sets. Ideally the user should be able to specify for each problem the number of edge sets and the names of each of them. Specifying the name of a new edge set alone should be enough for the user to be able to make a new type of connection between two nodes.

The user can make a timetable manually with the present implementation but the actual uses are rather limited. It might be useful to be able to create a timetable by hand and then specify that it or some other timetable is to be fixed before the algorithm is executed. That is to say that the students which are in these fixed timetables are removed from the routing and scheduling problem prior to executing the algorithm. In addition it might be useful to be able to edit a timetable generated by the algorithm.

Also it would probably be a good idea to allow the user to be able to interact with the algorithm in some way as it executes. When it seems that a non-optimal timetable would be generated, then the timetable as it has so far been generated should be displayed and the user be prompted to give a guiding hand to direct the algorithm in a certain direction. In a more general way it might be desirable to display each timetable as generated to check that it is acceptable. In [8] and [37] it is stated that an impediment to the acceptance of new computer systems is the feeling that the user has lost control of the problem. This can be partially overcome by incorporating a dynamic person-machine interaction. This would be both to

include implicit constraints which may be hard to define formally and to allow for expert human intuition to aid a heuristic algorithm.

The main improvement that might be made to the interface is to add a history menu. This would be a menu showing the changes that have been made to the network relative to the data file. Once a change was made initially in a graphical manner, the user would be able to undo or redo a change simply by selecting a menu option. Initially this might be just for changes to the physical network, though it might easily be augmented to include the problem parameters such as bus speeds and node demands. There could also be an additional menu to allow the user to change the parameters entered in the data file. At present these parameters can only be changed by evaluating text expressions. The statistics window would then have a complete listing of all changes so that the user can clearly identify which data pertain to which results.

Another change that might be made is to allow the user to magnify the whole network at once and then scroll the window over the magnified network. This would provide an alternate method of seeing more detail than the present method of magnifying only a limited area of the network.

The timetables for the St. Mark's problem are only preliminary but at present they are not as good as the other timetables. This observation is based on seeing many nodes in the timetables where no students are picked up and that most buses complete their routes with a high proportion of unfilled seats. One improvement that might be made is to change the algorithm where the network has a mixture of sparsely populated areas and densely populated areas. One possibility would be to have distinct sets of nodes as well as edges. Then each isolated group of village edges would be treated separately in a route first-cluster second manner.

Finally of course the algorithm could be made more general by relaxing some of the assumptions. There could be more than one school considered at once. Then buses might carry students from more than one school at once. It might be useful to explicitly consider the origins of the buses even in the single school case. The speeds of the buses might vary depending on their type. The students could be

classified into distinct sets so that each set could have attributes with different values. These values might include having different maximum allowable riding times. Also afternoon timetables might be considered where the travel times are asymmetric. There could be explicit time costs to allow for the boarding of students on the bus and for off-loading. Also routes may consider having an additional cost where the bus must turn around since this may be a slow manoeuvre. The implementation might also consider some of the other algorithms mentioned in the introduction so that the user could try more than one approach to solving the problem.

Bibliography

- [1] A.V. AHO, J.E. HOPCROFT and J.D. ULLMAN (1983). *Data Structures and Algorithms*, Addison-Wesley.
- [2] S. AKL (1980). *The minimal directed spanning graph for combinatorial optimization*, Australian Computer Journal, 12(4), 132-136.
- [3] R.D. ANGEL, W.L. CAUDLE, R. NOONAN and A. WHINSTON (1972). *Computer-assisted school bus scheduling*, Management Science, 18(6), B-279-B-288.
- [4] M. BALL, B. GOLDEN, A. ASSAD and L. BODIN (1983). *Planning for truck fleet size in the presence of a common carrier option*, Decision Sciences, 14(1), 103-120.
- [5] B.T. BENNETT and D.C. GAZIS (1972). *School bus routing by computer*, Transportation Research, 6(4), 317-325
- [6] L.D. BODIN and L. BERMAN (1979). *Routing and scheduling of school buses by computer*, Transportation Science, 13(2), 113-129.
- [7] L. BODIN and B. GOLDEN (1981). *Classification in vehicle routing and scheduling*, Networks, 11(2), 97-108.
- [8] L.D. BODIN, B.L. GOLDEN, A.A. ASSAD and M O. BALL (1983). *Routing and scheduling of vehicles and crews*, Computers and Operations Research, 10(2), 63-211.

- [9] L.D. BODIN and T. SEXTON (1979). *The subscriber dial-a-ride problem*, College of Business and Management, University of Maryland, College Park, Maryland Report No. UMCP-UMTA-1-79.
- [10] Carleton Roman Catholic School Board (1988). Revised resolution on transportation, section E.
- [11] Carleton Roman Catholic School Board memorandum (1981), re policy F23:03.
- [12] Miscellaneous documents from Carleton Roman Catholic School Board.
- [13] N. CHRISTOFIDES (1976). *Worst case analysis of a new heuristic for the traveling salesman problem*, Report 388, Graduate School of Industrial Administration, Carnegie Mellon University.
- [14] G. CLARKE and J. WRIGHT (1964). *Scheduling of vehicles from a central depot to a number of delivery points*, Operations Research, 12, 568-581.
- [15] S. EILON, C. WATSON-GANDY and N. CHRISTOFIDES (1971). *Distribution Management: Mathematical Modeling and Practical Analysis*. Hafner, New York.
- [16] G. GALLO and S. PALLOTTINO (1986). *Shortest path methods: a unifying approach*, Mathematical Programming Study, 26, 38-64.
- [17] M.R. GAREY and D.S. JOHNSON (1979). *Computers and Intractability: a Guide to the Theory of NP-Completeness*, Freeman, San Francisco.
- [18] B.L. GOLDEN (1977). *A statistical approach to the TSP*, Networks, 7, 209-225.
- [19] B.L. GOLDEN and A.A. ASSAD eds. (1988). *Vehicle Routing: Methods and Studies*, Studies in Management Science and Systems, Vol. 16, North Holland, Amsterdam.
- [20] B.L. GOLDEN and L.D. BODIN eds. (1981). *Proceedings of the international workshop on current and future directions in the routing and scheduling of vehicles and crews*, Networks, 11(2).

- [21] B.L. GOLDEN and W. STEWART (1981). *The empirical analysis of TSP heuristics*, Management Science & Statistics Working Paper No. 81-040, University of Maryland at College Park.
- [22] D. GRAHAM and H.L.W. NUTTLE (1986). *A comparison of heuristics for a school bus scheduling problem*, Transportation Research B, 20(2), 175-182.
- [23] J. HINSON and S. MULHERKAR (1975). *Improvements to the Clarke and Wright algorithm as applied to an airline scheduling problem*, Technical Report, Federal Express Corp.
- [24] O.H. IBARRA and C.E. KIM (1975). *Fast approximation algorithms for the knapsack and sum of subsets problems*, Journal of the Association for Computing Machinery, 22, 463-468.
- [25] C. KIM (1975). *A minimal spanning tree and approximate tours for a traveling salesman*, Comp. Sci. Tech. Report, University of Maryland.
- [26] P. KROLAK, W. FELTS and J. NELSON (1972). *A man-machine approach toward solving the generalized truck dispatching problem*, Transportation Science, 6, 149-170.
- [27] J.K. LENSTRA and A.H.G. RINNOOY KAN (1981). *Complexity of vehicle routing and scheduling problems*, Networks, 11(2), 221-227.
- [28] S. LIN (1965). *Computer solutions of the traveling salesman problem*, Bell System Technical Journal, 44, 2245-2269.
- [29] S. LIN and B. KERNIGHAN (1973). *An effective heuristic algorithm for the traveling salesman problem*, Operations Research, 21, 498-516.
- [30] W. LINDENBERG and F. FIALA (1982). *A school bus routing and scheduling problem*, Technical Report SCS-TR-5, School of Computer Science, Carleton University.

- [31] R.M. NEWTON (1970). *Bus routing in a multi-school system*, Ph.D. dissertation, State University of New York at Buffalo.
- [32] R.M. NEWTON and W.H. THOMAS (1969). *Design of school bus routes by computer*, Socio-Economic Planning Sciences, **3**, 75-85.
- [33] R.M. NEWTON and W.H. THOMAS (1974). *Bus routing in a multi-school system*, Computers and Operations Research, **1**, 213-222
- [34] J. NORBACK and R. LOVE (1977). *Geometric approaches to solving the traveling salesman problem*, Management Science, **23**, 1208-1223.
- [35] C. ORLOFF (1976). *Route constrained fleet scheduling*, Transportation Science, **10**, 149-168.
- [36] D.J. ROSENKRANTZ, R.E. STEARNS and P.M. LEWIS II (1977). *An analysis of several heuristics for the traveling salesman problem*, SIAM Journal on Computing, **6**(3), 563-581.
- [37] M.W.P. SAVELSBERGH (1988). *Computer aided routing*, Ph.D. thesis, Erasmus University, Amsterdam.
- [38] A.J. SWERSEY and W. EALLARD (1984). *Scheduling school buses*, Management Science, **30**(7), 844-857

Appendix A

Code for Routing and Scheduling

BusRoute Methods

optimumBuses

```
"The main heuristic procedure to find the optimum number
of buses to serve the network."
    | branchPoint spt startTime milliTime executionTimes innerTimetables
    bus outerTimetable aWStream |
CursorManager execute change.
"Collect partial execution times as we go."
startTime := Time millisecondClockValue.
executionTimes := OrderedCollection new: 20.
"Reset the network, buses, etc for new calculation."
self reset.
CursorManager normal change.
milliTime := Time millisecondClockValue.
"Find shortest path tree and all pairs shortest paths."
spt := self shortestPathTree.
CursorManager execute change.
spt restore.
executionTimes add:
    (Association key: 'spt' value: Time millisecondClockValue - milliTime).
milliTime := Time millisecondClockValue.
net allPairsShortestPaths.
executionTimes add:
    (Association key: 'Floyd' value: Time millisecondClockValue - milliTime).
milliTime := Time millisecondClockValue.

"main loop - generate timetables until outer students done."
[ net hasOuterStudents ] whileTrue: [
    branchPoint := spt furthestRemainingBranchPoint.
    executionTimes add:
        (Association key: 'branch'
            value: Time millisecondClockValue - milliTime).
    milliTime := Time millisecondClockValue.
    "depending on type of fork ..."
```

```

(spt isIsolatedFork: branchPoint)
"do knapsack route or..."
  ifTrue: [ outerTimetable := spt knapsack: branchPoint.
            (outerTimetable isKindOf: Timetable)
            ifTrue: [ self addTimetable: outerTimetable ]
            ifFalse: [ nonOptimumRoutes add: outerTimetable.
                      self addTimetable: (outerTimetable at: 1) ].
            executionTimes add:
              (Association key: 'knapsack'
                           value: Time millisecondClockValue - milliTime).
            milliTime := Time millisecondClockValue ]
"do detour route."
  ifFalse: [ outerTimetable := spt detour: branchPoint.
            (outerTimetable isKindOf: Timetable)
            ifTrue: [ self addTimetable: outerTimetable ]
            ifFalse: [ nonOptimumRoutes add: outerTimetable.
                      self addTimetable: (outerTimetable at: 1) ].
            executionTimes add:
              (Association key: 'detour'
                           value: Time millisecondClockValue - milliTime).
            milliTime := Time millisecondClockValue ]].

"do inner timetables."
innerTimetables := spt innerTimetables: self buses.
executionTimes add:
  (Association key: 'inner' value: Time millisecondClockValue - milliTime).
milliTime := Time millisecondClockValue.
"adjust outer timetables where bus is used on inner route."
innerTimetables do: [ :innerTimetable |
  bus := innerTimetable bus bus.
  outerTimetable := timetables detect: [ :t | t bus = bus ].
  timetables remove: outerTimetable.
  timetables add: (outerTimetable shift:
    (innerTimetable stops first time subtractTime:
     innerTimetable stops last time)).
  timetables add: innerTimetable ].
executionTimes add:
  (Association key: 'adjust' value: Time millisecondClockValue - milliTime).
milliTime := Time millisecondClockValue.
executionTimes addFirst:
  (Association key: 'total' value: Time millisecondClockValue - startTime).

"open window displaying number of buses used and execution times"
optString := String new.
aWStream := WriteStream on: optString.
aWStream cr;
  show: 'The approximate optimum number of buses is'; cr;
  show: self buses size printString, ' (+ ';
  show: self innerBuses size printString, ') using heuristics.'; cr;
  show: 'Millisecond timings: '; cr.
executionTimes do: [ :xTime |

```

```

aWStream show: ' ';
    show: (xTime key, ' ' copyFrom: 1 to: 16);
    show: (xTime value printPaddedTo: 6); cr ].
optString := aWStream contents.
clock time: net startingTime - net maximalRidingTime
            - net maximalWaitingTime.
CursorManager normal change.
self displayText: optString label: 'OptimumBuses'!

```

Network Methods

hasOuterStudents

```

"Answer true if there are nodes in the outer district
which still have students to be picked up."
^(shortestPathTree nodes
  detect: [ :node | node isOuterDistrict &
                  (node netNode currentDemand ~= 0) ]
  ifNone: [ nil ]) notNil!

```

Tree Methods

furthestRemainingBranchPoint

```

"Answer the branch point node furthest from the root,
where the fork still has outer
students to be picked up in the network. Give priority to branch
points in isolated components.
Answer the root if no other found."
| furthest nodes components componentNodes tNode |
"Find all branch points once only. Find branch points and sort them."
branchPoints isNil ifTrue: [
  branchPoints := SortedCollection sortBlock: [ :a :b |
    (self costToRoot: a) > (self costToRoot: b) ].
  self nodes do: [ :node |
    node isBranchPoint ifTrue: [ branchPoints add: node ].
  branchPoints := branchPoints asOrderedCollection.
  "Find branch points in small isolated components."
  components := network isolatedComponents.
  componentNodes := SortedCollection sortBlock: [ :a :b |
    (self costToRoot: a) > (self costToRoot: b) ].
  components do: [ :component |
    component do: [ :node |
      (tNode := self findTreeNode: node) isBranchPoint ifTrue: [
        componentNodes add: tNode ].
    componentNodes do: [ :node |
      branchPoints remove: node.
      branchPoints addFirst: node ].
  furthest := branchPoints first.
  furthest isRoot ifTrue: [ ~furthest ].
  (furthest descendants
    detect: [ :child | child isOuterDistrict &

```

```

                (child currentDemand ~= 0) ]
    ifNone: [ nil ]) notNil
    ifTrue: [ haltEnabled ifTrue: [ self halt ].
              ~furthest ]
    ifFalse: [ branchPoints remove: furthest.
               ~self furthestRemainingBranchPoint
               "check next node to see if children have
               already been routed in previous detours" ].!

```

Network Methods

isolatedComponents

```

"Answer the isolated components of the receiver
(apart from the largest one)."
| components |
components := self allIsolatedComponents.
components removeLast.
~components!

```

Network Methods

allIsolatedComponents

```

"Answer the isolated components of the receiver."
| nodeSet components nodes root |
nodes := self nodes.
root := shortestPathTree root.
components := SortedCollection sortBlock: [ :a :b | a size < b size ].
nodes remove: root.
"Find each component by growing a node-set depth first."
[ nodes notEmpty ] whileTrue: [
    nodeSet :=
        (self depthFirstGrow: nodes first
         root: root set: OrderedCollection new).
    nodeSet do: [ :node | nodes remove: node ].
    components add: nodeSet ].
~components!

```

Tree Methods

isIsolatedFork: branchPoint

```

"Answer true if the fork rooted at the branchPoint
in the network is isolated."
| descendants branchNodes |
branchNodes := OrderedCollection new.
branchPoint isBranchPoint | branchPoint isRoot
    ifFalse: [ self error: branchPoint printString,
                        ' is not a root of a fork ' ].
descendants := branchPoint descendants.
"Check each descendant of the branch point."
descendants do: [ :child |
    (child isLeaf and: [
        (network degreeOf: child netNode) ~= 1])

```

```

    ifTrue: [ ^false ]
    ifFalse: [ (child isInnerPoint and: [
        (network degreeOf: child netNode) ~= 2])
        ifTrue: [ ^false ]
        ifFalse: [ child isBranchPoint
            ifTrue: [ branchNodes add: child netNode ]]]].
"Check any sub-branch points."
branchNodes isEmpty ifTrue: [ ^true ].
descendants := descendants collect: [ :treeNode| treeNode netNode ].
branchNodes do: [ :node |
    ((network neighboursOf: node)
        detect: [ :neighbour | (descendants includes: neighbour) not ]
        ifNone: [ nil ])
        isKindOf: NetworkNode)
    ifTrue: [ ^false ]].
^true!

```

Tree Methods

knapsack: branchPoint

```

"Solve the knapsack problem with respect to time for
the isolated fork rooted at branchPoint."
| furthestNode costBranch limit descendants
  knapsack solutions solution length directDemand
  demand bus busA busFleet |
"length and limit are times"
busFleet := busRoute busFleet.
furthestNode := self furthestRemainingNode: branchPoint.
directDemand := (self directDemand: furthestNode) -
    (self directDemand: branchPoint).
"If the current demand of the longest branch is greater than the
maximum capacity bus, make timetable now."
(directDemand >= busFleet maxCapacity) ifTrue: [
    bus := busFleet capacity: busFleet maxCapacity.
    ^self makeTimetable: furthestNode branchPoint: branchPoint
    knapsack: OrderedCollection new
    startTime: startingTime - (self costToRoot: furthestNode)
    bus: bus].
costBranch := self costToRoot: branchPoint.
limit := maximalRidingTime - (self costToRoot: furthestNode) // 2.
"limit for knapsack problem"
"Create knapsack problem by adding nodes which are currently on the
end of isolated branches, except the longest branch."
descendants := branchPoint descendants.
descendants remove: furthestNode ifAbsent: [ ].
furthestNode ancestors do: [ :ancestor |
    descendants remove: ancestor ifAbsent: [ ]].
knapsack := OrderedCollection new.
descendants do: [ :descendant |

```

```

(descendant isCurrentLeaf) ifTrue: [
    knapsack add: (Association key: descendant
                        value: (self costToRoot: descendant) - costBranch)]]].
(knapsack size > 6) ifTrue: [
    Menu message: 'Large knapsack: ', knapsack size printString].
solutions := Enumeration new: knapsack
                    condition: [ :sol | sol last <= limit ]
                    zero: (Time fromSeconds: 0).

```

```

bus := nil.
[ bus isNil ] whileTrue: [
    "find a knapsack solution which is feasible wrt capacity"
    solution := solutions next.
    length := solution removeLast.
    demand := (solution inject: 0 into: [ :sum :assoc |
                        sum := sum + assoc key currentDemand ]) +
                directDemand.
    bus := busFleet capacity: demand.
    (bus notNil) ifTrue: [
        "Try to use nodes between branchPoint and the root"
        busA := busFleet capacity:
            (demand + (self directOuterDemand: branchPoint)
                    min: busFleet maxCapacity).
        (busA notNil) ifTrue: [ bus := busA ]]].
"Make knapsack route"
~self makeTimetable: furthestNode
    branchPoint: branchPoint
    knapsack: solution
    startTime: startingTime - (self costToRoot: furthestNode)
                        - (2 * length)
    bus: bus!

```

Tree Methods

```

makeTimetable: furthestNode
branchPoint: branchPoint
knapsack: solution
startTime: startTime
bus: bus
    "Private -
    Make a new timetable starting at furthestNode using the knapsack
    solution from the branchPoint.
    If there is an optimal timetable answer it,
    else answer a non optimal timetable with its defect."
    | time node furthestLeaf leafTime timetable parent leaves
      students nonOptimumTimetable |
    furthestLeaf := furthestNode.
    time := startTime.
    timetable := Timetable new name: busRoute nextTimetableName.
    timetable bus: bus.

```



```

"find all the leaves"
leaves := (solution collect: [ :assoc | assoc key ])
          asSortedCollection: [ :a :b |
            a currentDemand <= b currentDemand ].
"visit the leaf of each isolated branch"
[ furthestLeaf notNil ] whileTrue: [
  node := furthestLeaf.
  parent := node parent.
  "travel from leaf to branch point"
  [ node == branchPoint ] whileFalse: [
    (nodesSplittable | (node currentDemand <= bus emptySeats))
    ifTrue: [ students := node currentDemand min: bus emptySeats ]
    ifFalse: [ students := 0 ].
    timetable addTime: time
              place: node netNode
              students: students.
  "For nodes not on the longest branch, add the time on passing
  through the node on the way to the leaf."
  (node == furthestLeaf) | (furthestLeaf == furthestNode)
  ifTrue: [ leafTime := time ]
  ifFalse: [ timetable addTime:
              (leafTime subtractTime: (time subtractTime: leafTime))
              place: node netNode
              students: 0 ].
  time := time addTime: (network timeFrom: node netNode
                           to: parent netNode).
  node := parent.
  parent := node parent.
].
furthestLeaf := leaves isEmpty
              ifTrue: [ nil ]
              ifFalse: [ leaves removeFirst ].
(furthestLeaf notNil) ifTrue: [
  "add time for passing through branch point"
  timetable addTime: time
                place: branchPoint
                students: 0.
  "change time to time at next leaf"
  time := time + (self costToRoot: furthestLeaf) -
              (self costToRoot: branchPoint)].
].
node := branchPoint.
parent := node parent.
"visit nodes from branch point to school and try to fill the bus"
[ node notNil ] whileTrue: [
  (nodesSplittable | (node currentDemand <= bus emptySeats))
  ifTrue: [ students := node currentDemand min: bus emptySeats ]
  ifFalse: [ students := 0 ].
  timetable addTime: time
                place: node
                students: students.

```

```

(node notRoot) ifTrue: [
    time := time addTime: (network timeFrom: node netNode
                                to: parent netNode)].

node := parent.
(node notNil) ifTrue: [ parent := node parent ].
].

students := 0.
timetable stops do: [ :stop |
    (self findTreeNode: stop place) isOuterDistrict ifTrue: [
        students := students + stop passengers ]].
students < busRoute busFleet maxCapacity ifTrue: [
    nonOptimumTimetable := (Array with: timetable
                                with: busRoute busFleet maxCapacity -
                                students) ].

busRoute busFleet remove: bus.
haltEnabled ifTrue: [ self halt ].
nonOptimumTimetable isNil
    ifTrue: [ ^timetable ]
    ifFalse: [ ^nonOptimumTimetable ]!

```

Tree Methods

detour: branchPoint

```

"Solve a non-knapsack problem. Consider direct route and
then try different types of detours.
If an optimal timetable can be found return it,
else answer a nonoptimal timetable with the defect."
| startNode timetable demand detour7 detour7Data detourZigzagData
  fork directOuterDemand bus splitNodes nonOptimalTimetable |
startNode := self furthestRemainingNode: branchPoint.
"Try direct route only."
(,directOuterDemand := (self directOuterDemand: startNode))
> busRoute busFleet maxCapacity
ifTrue: [ timetable := Timetable new name: busRoute nextTimetableName.
    bus := busRoute busFleet capacity:
        busRoute busFleet maxCapacity.
    timetable bus: bus.
    busRoute busFleet remove: bus.
    self simpleRoute: startNode timetable: timetable.
    ^timetable ].
"Find number of students left in fork up to max bus capacity"
fork := branchPoint descendants.
demand := branchPoint isOuterDistrict
    ifTrue: [ branchPoint currentDemand ]
    ifFalse: [ 0 ].
fork do: [ :child |
    child isOuterDistrict ifTrue: [
        demand := demand + child currentDemand ]].
demand := demand min: busRoute busFleet maxCapacity.

```

```

"Try different types of detours."
detour7Data := self detour7: startNode.
detour7 := self shorten7Path: (detour7Data at: 2)
                direct: (self pathFrom: startNode).
splitNodes := OrderedCollection new.
splitNodes add: (detour7Data at: 3).
"If bus can be filled make detour 7 timetable..."
(detour7Data at: 1) + directOuterDemand =
    busRoute busFleet maxCapacity ifTrue: [
        timetable := self constructFromPath: detour7
                        splitNodes: splitNodes.
        haltEnabled ifTrue: [ self halt ].
        ^timetable ].
"else use zigzag detour route."
detourZigzagData := self detourZigzag: detour7
                    demand: (detour7Data at: 1) + directOuterDemand.
splitNodes addAll: (detourZigzagData at: 3).
timetable := self constructFromPath: (detourZigzagData at: 2)
                    splitNodes: splitNodes.
(timetable bus capacity >= busRoute busFleet maxCapacity) &
(timetable bus emptySeats = 0) ifTrue: [
    haltEnabled ifTrue: [ self halt ].
    ^timetable ].
nonOptimalTimetable := (Array with: timetable
    with: ((timetable bus capacity max: busRoute busFleet maxCapacity) -
        (detourZigzagData at: 1) - (detour7Data at: 1) -
        directOuterDemand)).
haltEnabled ifTrue: [ self halt ].
^nonOptimalTimetable!

```

Tree Methods

detour7: startNode

```

"Make a type 7 detour with startNode (a TreeNode) the start
of the direct part of the route. Find the one with the most
outer students whether or not the start of the detour is outside
the fork. Answer the number of students and the detour path. The
last node of the path is the one before the start of the direct path.
A type 7 detour is a direct route with additional stops before the
first stop on the direct part of the route."
| maxTime reachableNodes startNo nodeNo path students
  testStudents testPath testSplit directNodes restCapacity
  splitNode splitDemand |
startNo := startNode number.
maxTime := maximalRidingTime - (self costToRoot: startNode).
reachableNodes := OrderedCollection new.
"Reachable nodes are those in the outer district with students
remaining and can be reached within the time limit. They should
not already be part of the direct route"
self nodes do: [ :node |
    (node isOuterDistrict & (node currentDemand > 0) and: [
        (network costArray

```

```

        costFrom: (nodeNo := node number)
        to: startNo)
    <= maxTime ]) ifTrue: [
        reachableNodes add: nodeNo ]].
directNodes := (self pathFrom: startNode).
directNodes do: [ :dnode |
    reachableNodes remove: dnode number ifAbsent: [ ]].
students := 0.
path := OrderedCollection new.
restCapacity := busRoute busFleet maxCapacity -
    (self directOuterDemand: startNode).
splitNode := splitDemand := nil.
"Select the detour path where the number of students to be picked
up in the outer district excluding the direct part of the route is
the greatest provided all the students the students of the direct
part can be picked up."
reachableNodes do: [ :nodeNo |
    testPath := (network pathArray pathFrom: nodeNo to: startNo)
        collect: [ :i | self nodeNumber: i ].
    testPath removeLast.
    "disallow routes passing through school if these are not allowed"
    (mustDropAtSchool and: [ testPath includes: self root]) ifFalse: [
        testStudents := 0.
        testSplit := Array new: 2.
        testPath do: [ :pnode |
            (pnode isOuterDistrict & (testStudents < restCapacity)
                and: [ (directNodes includes: pnode) not ])
                ifTrue: [ (testStudents + pnode currentDemand > restCapacity)
                    ifTrue: [ nodesSplittable ifTrue: [
                        testSplit at: 1 put: pnode.
                        testSplit at: 2
                            put: restCapacity - testStudents.
                            testStudents := restCapacity ]]
                    ifFalse: [ testStudents := testStudents +
                        pnode currentDemand ]]].
            ((testStudents > students) or: [
                (testStudents = students) & (testSplit at: 1) isNil ])
                "better to have a non split node at start of detour"
                ifTrue: [ path := testPath.
                    splitNode := testSplit at: 1.
                    splitDemand := testSplit at: 2.
                    students := testStudents ]]].
    haltEnabled ifTrue: [ self halt ].
    ^Array with: students
        with: path
        with: (Association key: splitNode value: splitDemand)!

```

Tree Methods

detourZigzag: detour7 demand: detour7Demand

```

"Construct a zigzag type detour from the type 7 detour. The type 7
detour does not provide enough students to fill the largest remaining
bus so try to lengthen the present route to pick up more students.
The type 7 detour only has a demand of detour7Demand."
| detour7Cost oldNode limit reachableNodes treeNode restCapacity
  detourNode cost path splitNodes detourZigzag root
  reachableNode students |
restCapacity := busRoute busFleet maxCapacity - detour7Demand.
oldNode := detour7 first.
detour7Cost := 0.
detour7 do: [ :node |
  detour7Cost := (network costFrom: oldNode netNode to: node netNode)
    + detour7Cost.
  oldNode := node ].
"2 is for zag off detour7 route and return"
limit := (maximalRidingTime - detour7Cost / 2) floor.
"The reachable nodes are those which are not already on the path, which
are in the outer district which have students to be picked up. Also it
must be possible to get to the node and return to the original path in
the remaining time. Whether routes can pass through the school and
whether nodes are spittable must be considered."
reachableNodes := Set new.
network nodes do: [ :netNode |
  treeNode := (self findTreeNode: netNode).
  (detour7 includes: treeNode) not &
  (treeNode currentDemand ~= 0) &
  treeNode isOuterDistrict ifTrue: [
    detour7 do: [ :node7 |
      (network costArray costFrom: netNode number to: node7 number)
        < limit ifTrue: [
          (mustDropAtSchool not or: [
            ((network pathArray pathFrom: netNode number
              to: node7 number)
              includes: self root number) not ]) ifTrue: [
            nodesSplittable | (restCapacity >= node currentDemand)
              ifTrue: [ reachableNodes add: treeNode ]]]]].
    reachableNodes := reachableNodes asSortedCollection: [ :a :b |
      a currentDemand > b currentDemand ].
    splitNodes := OrderedCollection new.
    detourZigzag := detour7.
    students := 0.
    "Try each node as a node to add to the route."
    [ reachableNodes notEmpty ] whileTrue: [
      reachableNode := reachableNodes removeFirst.
      detourNode := detourZigzag first.
      cost := Time fromSeconds: 9999.
      "find where to divert from the original path"
      detourZigzag do: [ :nodeZ |
        ((network costArray costFrom: nodeZ number
          to: reachableNode number) < cost and: [
          mustDropAtSchool not or: [

```

```

        ((network pathArray pathFrom: reachableNode number to: nodeZ number)
         includes: self root number) not ]]) ifTrue: [
            cost := network costArray costFrom: nodeZ number
                    to: reachableNode number.

            detourNode := nodeZ ]].
cost < limit ifTrue: [
    "can detour to reachable node"
    limit := limit - cost.
    reachableNode currentDemand > restCapacity
        ifTrue: [ splitNodes add: (Association key: reachableNode
                                                value: restCapacity).
                  students := students + restCapacity.
                  restCapacity := 0.
                  reachableNodes := SortedCollection new ]
        ifFalse: [ restCapacity := restCapacity - reachableNode currentDemand.
                  students := students + reachableNode currentDemand ].
    nodesSplittable ifFalse: [
        reachableNodes copy do: [ :rnode |
            rnode currentDemand > restCapacity ifTrue: [
                reachableNodes remove: rnode ]]].

    "form extra path by starting at the detour point, proceeding the the
    node and returning on the same edges"
    path := network pathArray pathFrom: detourNode number
              to: reachableNode number.
    path removeFirst. "remove detourNode"
    path removeLast. "remove reachableNode"
    path addAll: (network pathArray pathFrom: reachableNode number
                        to: detourNode number).
    path removeLast. "remove detourNode"
    path := path collect: [ :i | self nodeNumber: i ].
    oldNode := detourZigzag findLast: [ :nodeZ | nodeZ == detourNode ].
    detourZigzag := (detourZigzag copyFrom: 1 to: oldNode),
                    path,
                    (detourZigzag copyFrom: oldNode to: detourZigzag size).
    ] "end new detour"
]. "end while detour is as long as possible"
haltEnabled ifTrue: [ self halt ].
~Array with: students
    with: detourZigzag
    with: splitNodes!

```

Tree Methods

```
innerTimetables: outerBuses
```

```

    "Construct the timetables for those students who remain
    in the inner district. Reuse the buses from the outer timetables."
    [ direct detour newNode oldNode
      time timetable timetables buses ]
    timetables := OrderedCollection new.
    buses := BusFleet new.
    outerBuses do: [ :bus | buses add: bus ].

```

```

"Form each inner timetable."
[ (newNode := self furthestRemainingNode: self root) notRoot ] whileTrue: [
    direct := OrderedCollection new.
    (self pathFrom: newNode) do: [ :node |
        direct addFirst: node ].
    time := maximalWaitingTime - (self costToRoot: newNode).
    oldNode := self root.
    detour := OrderedCollection new.
    "Collect students starting with last node of direct part."
    detour add: direct removeLast.
    [ newNode ~= self root ] whileTrue: [
        oldNode := newNode.
        newNode := self ellipse: oldNode time: time.
        time := time - (network costFrom: oldNode netNode
                        to: newNode netNode).
        detour add: newNode ].
    timetable := (self constructInnerRoute: direct
                  detour: detour buses: buses).
    timetables add: timetable.
    buses remove: timetable bus bus
].
~timetables!

```

Appendix B

Input Data and Timetables for Gerolstein Problem

```
Smalltalk at: #GerolRoute put: nil.!\nGerolRoute := BusRoute new.\nTranscript cr; show: 'nodes '.\nGerolRoute addNode: (NetworkNode new name: 'Berlingen' position: '273 @ 827'\ndemand: 46).!\nGerolRoute addNode: (NetworkNode new name: 'Bewingen' position: '236 @ 840'\ndemand: 31).!\nGerolRoute addNode: (NetworkNode new name: 'Birresborn' position: '206 @ 758'\ndemand: 83).!\nGerolRoute addNode: (NetworkNode new name: 'Büschelch' position: '243 @ 779'\ndemand: 45).!\nGerolRoute addNode: (NetworkNode new name: 'Densborn' position: '190 @ 703'\ndemand: 81).!\nGerolRoute addNode: (NetworkNode new name: 'Dinkelseifen' position: '194 @ 815'\ndemand: 2).!\nGerolRoute addNode: (NetworkNode new name: 'Duppach' position: '167 @ 852'\ndemand: 24).!\nGerolRoute addNode: (NetworkNode new name: 'Essingen' position: '278 @ 847'\ndemand: 28).!\nGerolRoute addNode: (NetworkNode new name: 'Gees' position: '261 @ 805'\ndemand: 27).!\nGerolRoute addNode: (NetworkNode new name: 'Gerolstein' position: '235 @ 808'\ndemand: 0).!\nGerolRoute addNode: (NetworkNode new name: 'Hanert' position: '206 @ 733'\ndemand: 4).!\nGerolRoute addNode: (NetworkNode new name: 'Hinterhausen' position: '196 @ 798'\ndemand: 8).!\nGerolRoute addNode: (NetworkNode new name: 'Hohenfels' position: '288 @ 839'\ndemand: 9).!\nGerolRoute addNode: (NetworkNode new name: 'Kalenborn' position: '192 @ 850'\ndemand: 12).!
```


APPENDIX B. INPUT DATA AND TIMETABLES FOR GEROLSTEIN PROBLEM108

```
GerolRoute addNode: (NetworkNode new name: 'Kopp' position: '180 @ 761'
                        demand: 15)..!
GerolRoute addNode: (NetworkNode new name: 'Lissingen' position: '218 @ 803'
                        demand: 63)..!
GerolRoute addNode: (NetworkNode new name: 'Michelbach' position: '236 @ 769'
                        demand: 4)..!
GerolRoute addNode: (NetworkNode new name: 'Müllenborn' position: '194 @ 831'
                        demand: 69)..!
GerolRoute addNode: (NetworkNode new name: 'Mürlenbach' position: '188 @ 730'
                        demand: 56)..!
GerolRoute addNode: (NetworkNode new name: 'Niedereich' position: '235 @ 774'
                        demand: 3)..!
GerolRoute addNode: (NetworkNode new name: 'Oos' position: '177 @ 829'
                        demand: 57)..!
GerolRoute addNode: (NetworkNode new name: 'Pelm' position: '253 @ 821'
                        demand: 60)..!
GerolRoute addNode: (NetworkNode new name: 'Rockeskyll' position: '258 @ 845'
                        demand: 48)..!
GerolRoute addNode: (NetworkNode new name: 'Roth' position: '205 @ 844'
                        demand: 36)..!
GerolRoute addNode: (NetworkNode new name: 'Scheuern' position: '185 @ 852'
                        demand: 16)..!
GerolRoute addNode: (NetworkNode new name: 'Schwast' position: '206 @ 814'
                        demand: 1)..!
GerolRoute addNode: (NetworkNode new name: 'Weissenseifen' position: '148 @ 722'
                        demand: 3)..!
```

Transcript show: 'connections '.

```
GerolRoute connect: 'Berlingen' to: 'Bewingen' distance: 65;
connect: 'Berlingen' to: 'Hohenfels' distance: 20;
connect: 'Berlingen' to: 'Pelm' distance: 35;
connect: 'Berlingen' to: 'Rockeskyll' distance: 55.!
GerolRoute connect: 'Bewingen' to: 'Berlingen' distance: 65;
connect: 'Bewingen' to: 'Gerolstein' distance: 40;
connect: 'Bewingen' to: 'Pelm' distance: 40;
connect: 'Bewingen' to: 'Rockeskyll' distance: 60;
connect: 'Bewingen' to: 'Roth' distance: 65.!
GerolRoute connect: 'Birresborn' to: 'Kopp' distance: 40;
connect: 'Birresborn' to: 'Lissingen' distance: 55;
connect: 'Birresborn' to: 'Mürlenbach' distance: 40.!
GerolRoute connect: 'Büschelch' to: 'Gerolstein' distance: 50;
connect: 'Büschelch' to: 'Michelbach' distance: 40;
connect: 'Büschelch' to: 'Niedereich' distance: 20.!
GerolRoute connect: 'Densborn' to: 'Hanert' distance: 50;
connect: 'Densborn' to: 'Mürlenbach' distance: 30;
connect: 'Densborn' to: 'Weissenseifen' distance: 30.!
GerolRoute connect: 'Dinkelseifen' to: 'Lissingen' distance: 25.!
GerolRoute connect: 'Duppach' to: 'Oos' distance: 30;
connect: 'Duppach' to: 'Scheuern' distance: 20.!
GerolRoute connect: 'Essingen' to: 'Hohenfels' distance: 15;
connect: 'Essingen' to: 'Pelm' distance: 50;
```

APPENDIX B. INPUT DATA AND TIMETABLES FOR GEROLSTEIN PROBLEM109

```

connect: 'Essingen' to: 'Rockeskyll' distance: 40.!
GerolRoute connect: 'Gees' to: 'Gerolstein' distance: 30;
connect: 'Gees' to: 'Pelm' distance: 20.!
GerolRoute connect: 'Gerolstein' to: 'Bewingen' distance: 40;
connect: 'Gerolstein' to: 'Büschelch' distance: 50;
connect: 'Gerolstein' to: 'Gees' distance: 30;
connect: 'Gerolstein' to: 'Lissingen' distance: 10;
connect: 'Gerolstein' to: 'Pelm' distance: 25;
connect: 'Gerolstein' to: 'Roth' distance: 45.!
GerolRoute connect: 'Hanert' to: 'Densborn' distance: 50;
connect: 'Hanert' to: 'Mürtenbach' distance: 20.!
GerolRoute connect: 'Hinterhausen' to: 'Kopp' distance: 45;
connect: 'Hinterhausen' to: 'Lissingen' distance: 25.!
GerolRoute connect: 'Hohenfels' to: 'Berlingen' distance: 20;
connect: 'Hohenfels' to: 'Essingen' distance: 15.!
GerolRoute connect: 'Kalenborn' to: 'Müllenborn' distance: 20;
connect: 'Kalenborn' to: 'Roth' distance: 15;
connect: 'Kalenborn' to: 'Scheuern' distance: 10.!
GerolRoute connect: 'Kopp' to: 'Birresborn' distance: 40;
connect: 'Kopp' to: 'Hinterhausen' distance: 45.!
GerolRoute connect: 'Lissingen' to: 'Birresborn' distance: 55;
connect: 'Lissingen' to: 'Dinkelseifen' distance: 25;
connect: 'Lissingen' to: 'Gerolstein' distance: 10;
connect: 'Lissingen' to: 'Hinterhausen' distance: 25;
connect: 'Lissingen' to: 'Schwast' distance: 25.!
GerolRoute connect: 'Michelbach' to: 'Büschelch' distance: 40;
connect: 'Michelbach' to: 'Niedereich' distance: 70.!
GerolRoute connect: 'Müllenborn' to: 'Kalenborn' distance: 20;
connect: 'Müllenborn' to: 'Oos' distance: 25;
connect: 'Müllenborn' to: 'Roth' distance: 15;
connect: 'Müllenborn' to: 'Schwast' distance: 15.!
GerolRoute connect: 'Mürtenbach' to: 'Birresborn' distance: 40;
connect: 'Mürtenbach' to: 'Densborn' distance: 30;
connect: 'Mürtenbach' to: 'Hanert' distance: 20;
connect: 'Mürtenbach' to: 'Weissenseifen' distance: 50.!
GerolRoute connect: 'Niedereich' to: 'Büschelch' distance: 20;
connect: 'Niedereich' to: 'Michelbach' distance: 70.!
GerolRoute connect: 'Oos' to: 'Duppach' distance: 30;
connect: 'Oos' to: 'Müllenborn' distance: 25;
connect: 'Oos' to: 'Scheuern' distance: 35.!
GerolRoute connect: 'Pelm' to: 'Berlingen' distance: 35;
connect: 'Pelm' to: 'Bewingen' distance: 40;
connect: 'Pelm' to: 'Essingen' distance: 50;
connect: 'Pelm' to: 'Gees' distance: 20;
connect: 'Pelm' to: 'Gerolstein' distance: 25;
connect: 'Pelm' to: 'Rockeskyll' distance: 30.!
GerolRoute connect: 'Rockeskyll' to: 'Berlingen' distance: 55;
connect: 'Rockeskyll' to: 'Bewingen' distance: 60;
connect: 'Rockeskyll' to: 'Essingen' distance: 40;
connect: 'Rockeskyll' to: 'Pelm' distance: 30.!
GerolRoute connect: 'Roth' to: 'Bewingen' distance: 65;

```

APPENDIX B. INPUT DATA AND TIMETABLES FOR GEROLSTEIN PROBLEM110

```
connect: 'Roth' to: 'Gerolstein' distance: 45;
connect: 'Roth' to: 'Kalenborn' distance: 15;
connect: 'Roth' to: 'Müllenborn' distance: 15.!
GerolRoute connect: 'Scheuern' to: 'Duppach' distance: 20;
connect: 'Scheuern' to: 'Kalenborn' distance: 10;
connect: 'Scheuern' to: 'Oos' distance: 35.!
GerolRoute connect: 'Schwast' to: 'Lissingen' distance: 25;
connect: 'Schwast' to: 'Müllenborn' distance: 15.!
GerolRoute connect: 'Weissenseifen' to: 'Densborn' distance: 30;
connect: 'Weissenseifen' to: 'Mürtenbach' distance: 50.!
```

```
Transcript show: 'parameters'; cr.
GerolRoute startingTime: (Time fromString: '08:15:00').
GerolRoute maximalRidingTime: (Time fromString: '00:45:00').
GerolRoute maximalWaitingTime: (Time fromString: '00:15:00').
GerolRoute averageBusSpeed: 350.
GerolRoute haltEnabled: false.
GerolRoute nodesSplittable: true.
GerolRoute mustDropAtSchool: false.!
```

```
Transcript show: 'buses '.
```

```
GerolRoute busFleet
add: (Bus new capacity: 98; name: 'Bus1');
add: (Bus new capacity: 98; name: 'Bus2');
add: (Bus new capacity: 48; name: 'Bus3');
add: (Bus new capacity: 48; name: 'Bus4');
add: (Bus new capacity: 48; name: 'Bus5');
add: (Bus new capacity: 45; name: 'Bus6');
add: (Bus new capacity: 45; name: 'Bus7');
add: (Bus new capacity: 45; name: 'Bus8');
add: (Bus new capacity: 36; name: 'Bus9');
add: (Bus new capacity: 36; name: 'Bus10');
add: (Bus new capacity: 36; name: 'Bus11');
add: (Bus new capacity: 27; name: 'Bus12');
add: (Bus new capacity: 27; name: 'Bus13');
add: (Bus new capacity: 27; name: 'Bus14');
add: (Bus new capacity: 27; name: 'Bus15');
add: (Bus new capacity: 27; name: 'Bus16');
add: (Bus new capacity: 21; name: 'Bus17');
add: (Bus new capacity: 21; name: 'Bus18');
add: (Bus new capacity: 18; name: 'Bus19').!
```

APPENDIX B. INPUT DATA AND TIMETABLES FOR GEROLSTEIN PROBLEM111

B.1 Gerolstein Timetables

Timetable: Timetable2	Bus: Bus1	Capacity: 98
Oos	07:25:17	57
Müllenborn	07:29:34	38
Schwast	07:32:08	0
Lissingen	07:36:25	0
Gerolstein	07:38:08	0
Bütscheich	07:46:42	0
Niedereich	07:50:08	3
Bütscheich	07:53:34	0
Gerolstein	08:02:08	0

Timetable: Timetable1	Bus: Bus2	Capacity: 98
Michelbach	07:25:17	4
Bütscheich	07:32:08	45
Gerolstein	07:40:42	0
Lissingen	07:42:25	0
Birresborn	07:51:51	49
Lissingen	08:01:17	0
Gerolstein	08:03:00	0

Timetable: Timetable10	Bus: Bus9	Capacity: 36
Duppach	07:25:18	7
Scheuern	07:28:44	0
Kalenborn	07:30:27	0
Roth	07:33:01	27
Gerolstein	07:40:44	0
Pelm	07:45:01	0
Berlingen	07:51:01	2
Pelm	07:57:01	0
Gerolstein	08:01:18	0

Timetable: Timetable8	Bus: Bus6	Capacity: 45
Duppach	07:33:52	17
Scheuern	07:37:18	16
Kalenborn	07:39:01	12
Roth	07:41:35	0
Gerolstein	07:49:18	0
Pelm	07:53:35	0
Essingen	08:02:09	0
Pelm	08:10:43	0
Gerolstein	08:15:00	0

Timetable: Timetable6	Bus: Bus8	Capacity: 45
Hanert	07:39:52	4
Mürtenbach	07:43:18	0
Birresborn	07:50:09	0
Kopp	07:57:00	11
Birresborn	08:03:51	30

APPENDIX B. INPUT DATA AND TIMETABLES FOR GEROLSTEIN PROBLEM112

Lissingen	08:13:17	0
Gerolstein	08:15:00	0

Timetable: Timetable13	Bus: Bus15	Capacity: 27
Kopp	07:42:27	4
Hinterhausen	07:50:10	0
Lissingen	07:54:27	0
Schwast	07:58:44	0
Müllenborn	08:01:18	0
Roth	08:03:52	9
Müllenborn	08:06:26	14
Schwast	08:09:00	0
Lissingen	08:13:17	0
Gerolstein	08:15:00	0

Timetable: Timetable12	Bus: Bus12	Capacity: 27
Rockeskyll	07:48:26	12
Pelm	07:53:35	0
Gerolstein	07:57:52	0
Lissingen	07:59:35	0
Schwast	08:03:52	0
Müllenborn	08:06:26	15
Schwast	08:09:00	0
Lissingen	08:13:17	0
Gerolstein	08:15:00	0

Timetable: Timetable3	Bus: Bus5	Capacity: 48
Weissenstein	07:48:26	3
Mürtenbach	07:57:00	45
Birresborn	08:03:51	0
Lissingen	08:13:17	0
Gerolstein	08:15:00	0

Timetable: Timetable5	Bus: Bus4	Capacity: 48
Densborn	07:51:51	33
Mürtenbach	07:57:00	11
Birresborn	08:03:51	4
Lissingen	08:13:17	0
Gerolstein	08:15:00	0

Timetable: Timetable4	Bus: Bus3	Capacity: 48
Densborn	07:51:51	48
Mürtenbach	07:57:00	0
Birresborn	08:03:51	0
Lissingen	08:13:17	0
Gerolstein	08:15:00	0

Timetable: Timetable14	Bus: Bus19	Capacity: 18
Müllenborn	07:54:26	2
Schwast	07:57:00	1
Lissingen	08:01:17	15

APPENDIX B. INPUT DATA AND TIMETABLES FOR GEROLSTEIN PROBLEM113

Gerolstein 08:03:00 0

Timetable: Timetable9	Bus: Bus11	Capacity: 36
Berlingen	07:56:09	8
Hohenfels	07:59:35	0
Essingen	08:02:09	28
Pelm	08:10:43	0
Gerolstein	08:15:00	0

Timetable: Timetable7	Bus: Bus7	Capacity: 45
Hohenfels	08:01:17	9
Berlingen	08:04:43	36
Pelm	08:10:43	0
Gerolstein	08:15:00	0

Timetable: Timetable15	Bus: IBus9	Capacity: 36
Gerolstein	08:01:18	0
Bewingen	08:08:09	31
Gerolstein	08:15:00	0

Timetable: Timetable18	Bus: IBus1	Capacity: 98
Gerolstein	08:02:08	0
Gees	08:07:17	27
Pelm	08:10:43	60
Gerolstein	08:15:00	0

Timetable: Timetable16	Bus: IBus2	Capacity: 98
Gerolstein	08:03:00	0
Lissingen	08:04:43	0
Dinkelseifen	08:09:00	2
Lissingen	08:13:17	48
Gerolstein	08:15:00	0

Timetable: Timetable17	Bus: IBus19	Capacity: 18
Gerolstein	08:03:00	0
Lissingen	08:04:43	0
Hinterhausen	08:09:00	8
Lissingen	08:13:17	0
Gerolstein	08:15:00	0

Timetable: Timetable11	Bus: Bus10	Capacity: 36
Rockeskyll	08:05:34	36
Pelm	08:10:43	0
Gerolstein	08:15:00	0

END

1 2 0 4 9 1

FIN