RALF BORNDÖRFER MARTIN GRÖTSCHEL
FRIDOLIN KLOSTERMEIER CHRISTIAN KÜTTNER

Telebus Berlin:
Vehicle Scheduling in a Dial-a-Ride System

Preprint SC 97-23 (April 1997)

Telebus Berlin: Vehicle Scheduling in a Dial-a-Ride System*

Ralf Borndörfer[†] Martin Grötschel[†] Fridolin Klostermeier[‡] Christian Küttner[‡]

Abstract. *Telebus* is Berlin's dial-a-ride system for handicapped people that cannot use the public transportation system. The service is provided by a fleet of about 100 mini-busses and includes aid to get in and out of the vehicle. Telebus has between 1,000 and 1,500 transportation requests per day. The problem arises to schedule these requests into the vehicles such that punctual service is provided while operation costs should be minimum. Additional constraints include pre-rented vehicles, fixed bus driver shift lengths, obligatory breaks, and different vehicle capacities.

We use a *set partitioning* approach for the solution of the bus scheduling problem that consists of two steps. The first *clustering* step identifies segments of possible bus tours ("orders") such that more than one person is transported at a time; the aim in this step is to reduce the size of the problem and to make use of larger vehicle capacities. The problem to select a set of orders such that the traveling distance of the vehicles within the orders is minimal is a set partitioning problem that we can solve to optimality. In the second step the selected orders are *chained* to yield possible bus tours respecting all side constraints. The problem to select a set of such bus tours such that each order is serviced once and the total traveling distance of the vehicles is minimum is again a set partitioning problem that we solve approximately.

We have developed a computer system for the solution of the bus scheduling problem that includes a branch-and-cut algorithm for the solution of the set partitioning problems. A version of this system is in operation at Telebus since July 1995. Its use made it possible that Telebus can service today about 30% more requests per day for the same amount of money than before.

Keywords. Dial-a-Ride Systems, Set Partitioning

Mathematics Subject Classification (MSC 1991). 90B06, 90C10, 90B90

1 Handicapped People's Transport in Berlin

Better accessibility of the public transportation system has become an important political goal for many municipalities: They introduce low-floor busses, install lifts in subway stations, etc. But many handicapped and elderly people still have problems because they need additional help, the next station is too far away, or the line they want to use is not yet accessible. Berlin, like many other cities, offers to these people a special transportation service: The so-called *Telebus* provides a door-to-door transportation and aid at the pick-up and the target point. The system is financed by the Senate of Berlin's department for Social Affairs (SenSoz) and operated by the Berliner Zentralausschuß für Soziale Aufgaben e.V. (BZA), an association of charitable organizations. Figure 1 shows a Telebus picking up a customer.

Figure 1: A Telebus picks up a customer.

Telebus is a *dial-a-ride system*: Every entitled user (currently about 25,000 people) can order up to 50 rides per month at the BZA's telephone central.

*The Telebus-Project is supported by the Senate of Berlin's Department for Science, Research, and Culture.

[†]Konrad-Zuse-Zentrum für Informationstechnik Berlin, Takustraße 7, 14195 Berlin, Germany, Email: `[surname]@zib.de`, URL: `http://www.zib.de`

[‡]Intranetz Gesellschaft für Informationslogistik mbH, Klopstockstraße 9, 14163 Berlin, Email: `info@intranetz.de`, URL: `http://www.intranetz.de`

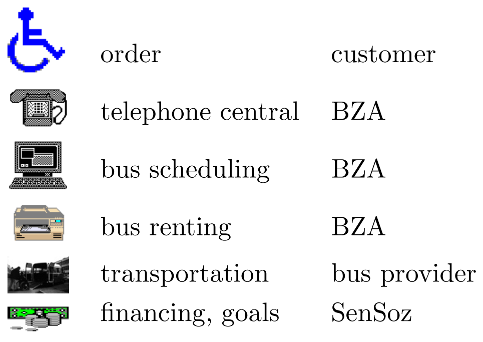


Figure 2: The Telebus System.

If the order is placed one day in advance, Telebus guarantees to service the ride as requested, later "spontaneous" requests are serviced as possible. The advance orders, about 1,500 during the week and 1,000 on weekends, are collected and scheduled into a fleet of mini-busses that are rented on demand from charitable organizations and commercial companies. These busses will pick-up the customers at the desired time (modulo a certain tolerance) and transport him/her to the target; if required, the crew will provide aid to leave the apartment, enter the vehicle, etc. This service is available every day from 5 am in the morning to 1 am at night. Figure 2 gives a diagram of the Telebus system.

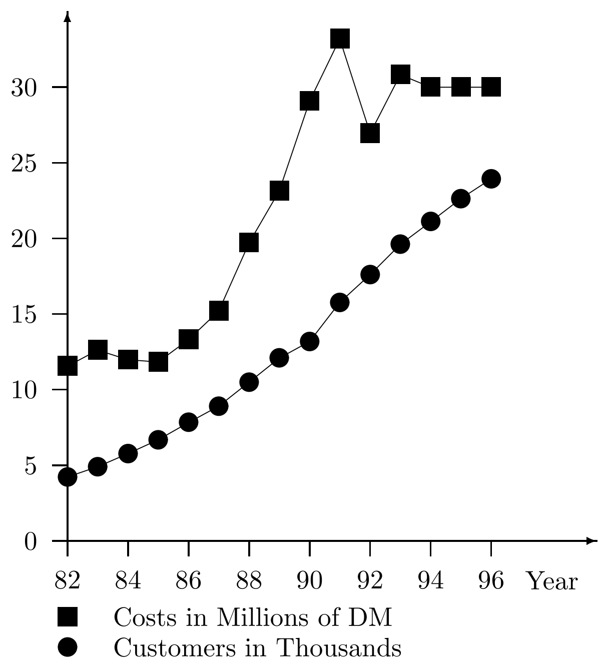


Figure 3: Development of Telebus.

Telebus was established 15 years ago and since then the number of customers and orders increased constantly. Until recently, the *vehicle scheduling* was done manually by experienced planners that could work out a feasible schedule in about 16 man-hours. But when East Berlin's handicapped people also started to use the system after the reunification of Germany, it was clear that the traditional way of scheduling would not be able to cope with the projected additional demand. And the problem was not only to come up with a *feasible* schedule: More requests in a doubled area of service lead to rising costs and put the system under a heavy pressure for optimization. Figure 3 gives an impression of these explosive developments; the numbers for the years up to 1993 are taken from T 336 of the report of Berlin's audit division for the year 1994, the other data was provided by the BZA.

Modern computer hard- and software was needed to solve the scheduling problems of the BZA and the *Telebus-project*, a cooperation between the Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB), the BZA, and the SenSoz, was started to develop it. The result of the project is a new *Telebus-computersystem*, that supports and integrates the complete sequence of operations at the BZA: Ordering, vehicle scheduling, radio telephony, accounting, controlling, and statistics. The system consists of a tool box of software modules, runs on a network of 20 MacIntosh PCs, and is in operation at the BZA since 1995. Its use, together with a simultaneous reorganization of many parts of the Telebus service, lead to

(i) improvements in service, for example, a reduction of the advance ordering period from three days in 1992 to one day today (needed for vehicle renting) or increased punctuality of the computer schedule in comparison to the result of manual planning,

(ii) cost reductions, such that today about 30% more requests can be serviced for the same money as in 1992, and

(iii) simplifications of the work in the Telebus central.

A comparison of the number of requests and the costs for a month in 1994 before and in 1996 after the installation of the system is shown in Figure 4.

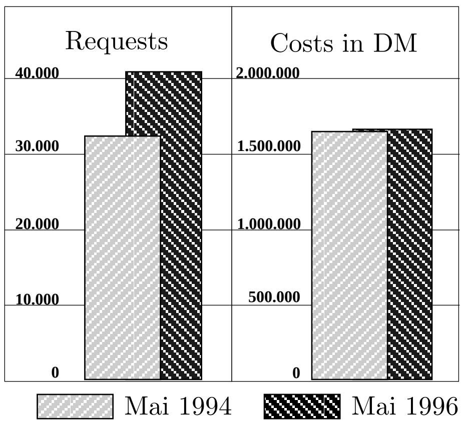


Figure 4: Results of the Telebus-project.

The heart of the Telebus-computersystem is the *vehicle scheduling module*. This module is based on mathematical optimization techniques that we want to describe in this article. Our aim is to show that methods of this kind can make a significant contribution to the solution of real world transportation problems: The results at Telebus are of interest for similar dial-a-ride systems. It goes without saying, however, that optimization at Telebus did not only consist of a better vehicle scheduling, but involved many other important factors: Restructuring of the operation of the central, negotiations with vehicle providers, and personal dedication (Ch. Küttner and F. Klostermeier, in particular, worked for more than a year in the Telebus central, drove on Telebusses, etc.). More details on this *consulting aspect* of the project can be found in Klostermeier and Küttner [1993] and Borndörfer, Grötschel, Herzog, Klostermeier, Konsek, and Küttner [1996] (both articles in German).

2 Vehicle Scheduling at Telebus

The most important task at Telebus is the daily construction of the *vehicle schedule*: The schedule determines both *operational costs* for vehicles and crews and *customer satisfaction* in terms of punctual service. The vehicle scheduling problem (VSP) at Telebus can be stated in an informal way as follows:

(VSP) Given a number of requests and a number of available vehicles, rent a suitable set of vehicles and schedule all requests into them such that a number of constraints like punctuality and labour regulations are satisfied and operational costs are minimum.

The aim of this section is to describe the VSP precisely and to introduce our set partitioning approach for its solution. We start with a discussion of the VSP's data, its constraints, and objectives.

The basis for vehicle scheduling are some number ν of *vehicles* of different types. Actually, a vehicle is in this context not only a car, but always comes complete with a crew for a shift of operation: The BZA does not rent vehicles, but shifts of operation of a car and a crew. Such a (manned) vehicle b, $b = 1, \ldots, \nu$, is characterized by the following data:

	(i)	c_b	type (class): Teletaxi, 1- and 2-bus small or large
(V)		$A_b = (A_b^{\text{w-chair}}, A_b^{\text{seat}})$	capacity: no. of wheelchair places and seats
	(ii)	g_b	group: type, depot location, shift

There are approximately $\nu = 100$ busses available for renting. Vehicles can be distinguished by a type (or class) and a group. There are five types: Teletaxis, small busses with one driver (1-bus), large 1-busses, small 2-busses, and large 2-busses. The type is important for deciding whether a request can be serviced by a particular vehicle: Teletaxis can transport only customers without or with folding wheelchairs, non-folding wheelchairs require a bus, and staircase aid a bus with a crew of two. The type of a vehicle determines also its capacity: Teletaxis can transport one handicapped customer and one non-handicapped companion, small busses have a capacity of $(2, 3)$, large busses of $(3, 4)$. Capacity is a sub-parameter of the type, but gets a symbol on its own for convenience of notation. Vehicles of the same type fall into groups, that play a role for the construction of tours: A group contains vehicles that are indistinguishable in the sense that they have the same type, are stationed at the same depot, and can be rented for identical shifts.

The vehicles will be used to service some number m of transportation *requests*. The following data is associated to each request $i = 1, \ldots, m$:

	(i)	$v_i^{\text{pick-up}}, v_i^{\text{target}}$	pick-up and target node
	(ii)	$p(v_i^{\text{pick-up}}), p(v_i^{\text{target}})$	pick-up and target point
	(iii)	$T(v_i^{\text{pick-up}}) = [\underline{t}(v_i^{\text{pick-up}}), \overline{t}(v_i^{\text{pick-up}})]$	interval of feasible times to arrive at pick-up point
(R)		$T(v_i^{\text{target}}) := [\underline{t}(v_i^{\text{target}}), \overline{t}(v_i^{\text{target}})]$	interval of feasible times to arrive at target point
	(iv)	$t^{\text{service}}(v_i^{\text{pick-up}}), t^{\text{service}}(v_i^{\text{target}})$	service time at pick-up and target point
	(v)	C_i	set of feasible vehicle types
	(vi)	$a_i = (a_i^{\text{w-chair}}, a_i^{\text{seat}})$	no. of wheelchairs and seats needed

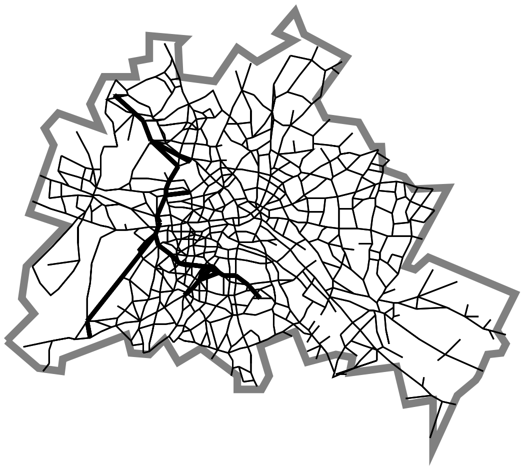


Figure 5: Graph of Berlin.

There is a *pick-up node* $v_i^{\text{pick-up}}$ and a *target node* v_i^{target}, that correspond to the pick-up and delivery *events* of a request. The pick-up and target locations or points $p(v_i^{\text{pick-up}})$ and $p(v_i^{\text{target}})$ of a request are stored as nodes of a graph of Berlin that is shown in Figure 5. The 2,510 edges of this graph are labelled with average travelling times and distances that we use to compute shortest routes between its 828 nodes. In addition to this spatial information, a request bears temporal data that is measured in units of 5 minutes. There is an interval of feasible pick-up times $T(v_i^{\text{pick-up}})$ that is computed according to Telebus specific rules. The rules try to find a compromise between punctual service and more degrees of freedom for the vehicle scheduling process. Currently, most requests have

$$T(v_i^{\text{pick-up}}) = t^\star(v_i^{\text{pick-up}}) + [-3, 3],$$

where $t^\star(v_i^{\text{pick-up}})$ is the time desired by the customer, i.e., the vehicle is allowed to arrive $3*5 = 15$ minutes early or late. Similar, but more complex rules are used to determine a feasible time interval $T(v_i^{\text{target}})$ to arrive at the target; here, the shortest possible travelling time and a maximum detour time play a role. Finally, some service time $t^{\text{service}}(v_i^{\text{pick-up}})$ and $t^{\text{service}}(v_i^{\text{target}})$ is needed at the pick-up and the target point. The amount of aid, the wheelchair, and other factors determine what kind of vehicles C_i (Teletaxi, 1-bus, or 2-bus) can or must be used, and the final load data a_i gives the number of wheelchair places and seats needed. An impression of the distribution of Teletaxi, 1-bus, and 2-bus can be obtained from Figure 6, that shows a typical statistic of this kind.

Rules for feasible vehicle tours arise primarily from bus renting contracts and labour regulations for bus drivers. Renting contracts are such that most vehicles are available for shifts of $8\frac{1}{2}$ or $10\frac{1}{2}$ hours, some others can be rented by the hour to cover peaks of demand. The majority of renting is done on a daily basis on demand, but vehicles can also be rented on a long-term basis. Labour regulations prescribe maximum driving hours and rules for obligatory breaks. The current rule at Telebus is that a break of 30 minutes has to be taken between the fourth and sixth hour of a shift. Two other rules state that a feasible vehicle tour must start and end at the vehicle's depot, and that it is not allowed to wait or make a break with a customer "on board".

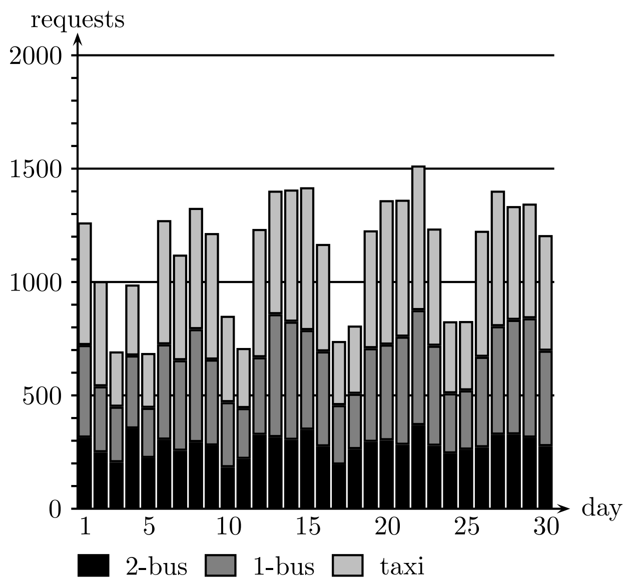
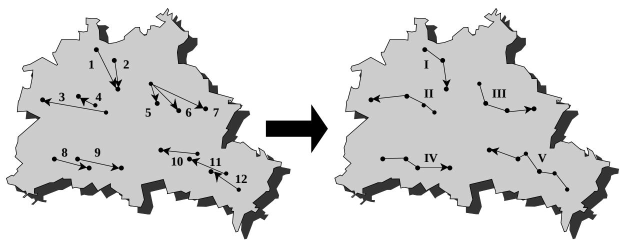


Figure 6: Telebus request pattern for june 1995.

The *objective* of the VSP is to minimize operational costs, but the BZA seldom use this criterion in its pure form. The reason is that the planned schedule and the one that is really executed on the next day differ significantly because of cancellations of requests, spontaneous requests, vehicle breakdowns, and other unpredictable events. The BZA must safeguard against every day's emergency situations and does so by preferring "safer" plans at some extra cost. The main tool to do this is to introduce components into the objective that aim at schedules of a safer type; we will come back to this point in the discussion of the set partitioning model.

Our solution approach for the VSP is based on the concept of a *cluster* of requests. A cluster or, in BZA terminology, an *order*, consists of a set of requests that are advantageously serviced in a simultaneous way. It corresponds to a maximal subtour such that the vehicle is never empty: The subtour starts with an empty vehicle picking up a first customer, services the requests of the cluster, and becomes empty for the first time when the last customer leaves the vehicle at his/her target. This results in a "simultaneous service" of the requests in the cluster in the sense that, while one customer is transported, at least one other person is picked up or transported to his/her target. Figure 7 shows a number of clusters: Collections, insertions, simple and continued concatenations.



I	collection with common target point
II	insertion
III	collection with common pick-up point
IV	concatenation
V	continued concatenation

Figure 7: Cluster types at Telebus.

Clusters can be used to *decompose* the vehicle scheduling process into two phases: A *clustering* phase that combines requests to clusters and a subsequent *chaining* phase that builds tours as sequences of clusters. The flavour of clustering is that of a local optimization to make use of larger vehicle capacities, while chaining must deal with constraints for the feasibility of complete tours, like depot locations, breaks, and shift lengths. The advantage of this approach is that it gets easier to construct tours from a comparably smaller number of orders in an nonoverlapping way, the disadvantage is that a hierarchical planning process in steps will generally yield suboptimal solutions.

To use this approach, it makes sense to describe a cluster c as follows:

	(i)	$S_c := (v^1, \ldots, v^l)$	sequence of serviced pick-up and target nodes
	(ii)	$T_c^{\text{pick-up}} := [\underline{t}_c^{\text{pick-up}}, \overline{t}_c^{\text{pick-up}}]$	interval of feasible times to arrive at first pick-up point
(C)		$T_c^{\text{target}} := [\underline{t}_c^{\text{target}}, \overline{t}_c^{\text{target}}]$	interval of feasible times to end service at last target point
		t_c	total time to service cluster completely
	(iii)	C_c	set of feasible vehicle types

The subtour corresponding to a cluster is given by a sequence of pick-up and target nodes S_c that will be serviced in this order. More precisely, if $v^j = v_i^{\text{pick-up}}$ is a pick-up node, the vehicle will drive to the corresponding location and pick-up the customers complete with service, if $v^j = v_i^{\text{target}}$ is a target node, the vehicle will go to the target location and service the customer. A cluster sequence S_c must, of course, satisfy a couple of constraints: The initial node v^1 must be a pick-up node, the terminal node v^l a target node, each node can appear at most once, each target node must be preceeded by the pick-up node of the same request and vice versa, and the sequence must describe simultaneous service, i.e., either there is only one request ($l = 2$) or there is another node between any request's pick-up and target node. An important observation is that the cluster sequence determines the operation of the vehicle completely: Since it is not allowed to wait with a customer "on board", the vehicle either drives to the next node or the crew provides service. This means that the total time t_c to service the cluster is constant and that the service of the complete cluster can be shifted as a block over some feasible interval of time. Thus, there is a maximal interval $T_c^{\text{pick-up}}$ of feasible times to arrive at the first pick-up node of the cluster and a corresponding interval of feasible end times, and these have the property

$$t_c = \underline{t}_c^{\text{target}} - \underline{t}_c^{\text{pick-up}} = \overline{t}_c^{\text{target}} - \overline{t}_c^{\text{pick-up}}.$$

The sequence of serviced requests determines also the possible types of vehicles C_c: These depend on the most "demanding" vehicle type of the requests and the maximum number of occupied wheelchair places and seats needed.

Vehicle tours are the last structure that misses, and just as a cluster can be described as a sequence of request nodes, a tour k can be seen as a sequence of clusters:

(T)	(i)	$S_k := (c^1, \ldots, c^l)$	sequence of serviced clusters
	(ii)	$T_k^{\text{pick-up}} := [\underline{t}_k^{\text{pick-up}}, \overline{t}_k^{\text{pick-up}}]$	interval of feasible times to start service of the first cluster
		$T_k^{\text{target}} := [\underline{t}_k^{\text{target}}, \overline{t}_k^{\text{target}}]$	interval of feasible times to end service of the last cluster
		t_k	total time to service tour completely
	(iii)	k	vehicle

A tour k consists of a sequence of clusters S_k that are serviced in the given order. To deal with depot locations, breaks, and shift lengths we also allow for additional pull-in, break, and pull-out clusters. Pull-in clusters will prescribe a starting location and time of a tour, break clusters an obligatory break between the fourth and sixth hour of service of a tour, and pull-out clusters model again depot locations and maximum shift lengths. Pull-in and pull-out clusters will fix the possible times to begin and end a tour, but we nevertheless introduce the time windows $T_k^{\text{pick-up}}$ and T_k^{target} for later use in our tour construction algorithm. Additional parameters give the total time to service a tour, i.e., the shift length, and the vehicle.

What is a good way to do the clustering? In principle, one would like to construct a set of clusters that will result in the construction of a good set of tours — later. We try to approximate this goal using secondary criteria like the *travelling distance* or the *time* within the clusters. We are thus lead to consider the *clustering problem* to construct a set of clusters, such that each request is contained in exactly one cluster and some objective, like the sum of the internal travelling distances, is minimal. Given a decision for a set of clusters, the *chaining problem* can be stated in a similar way. This time, we want to construct a set of tours, such that each cluster is serviced by exactly one tour, such that there are enough vehicles of the required types and groups, and such that operational costs or a similar objective becomes minimal.

Both questions can be modeled as a *set partitioning problem*

$$\text{(SPP)} \qquad \begin{array}{rcl} \min c^T x & & \\ Ax & = & \mathbb{1} \\ x & \in & \{0,1\}^n, \end{array}$$

where $A \in \{0,1\}^{m \times n}$ is a 0/1-matrix and $c \in \mathbb{R}_+^n$ is a positive cost function.

In the clustering case, row i of the matrix A corresponds to request i, and each column $A_{\cdot j}$ of A to a feasible cluster: The entry a_{ij} is equal to one if cluster j services request i and zero else, the objective c_j denotes, for example, the internal travelling distance or time within the cluster. Then, the feasible solutions x of the integer program (SPP) are in one-to-one correspondence to sets S of clusters such that each request is contained in exactly one cluster via the relation $x_j = 1 \iff j \in S$ and the optimum solution $x^\star$ of (SPP) corresponds to the best such combination.

In the chaining case, the rows correspond to the clusters selected in the clustering step, the columns to tours, and the objective to some cost criterion associated to a tour like, for example, operation costs. The only additional point to consider is that the model as just stated does not respect *vehicle availabilities*: The tour matrix A contains for each vehicle all possible tour-columns that this vehicle can service, and it is possible that a solution of (SPP) will use a vehicle more than once. Considerations to prevent this lead to additional constraints of the form

$$\sum_{j \in J(k)} x_j \leq 1 \quad \text{or} \quad \sum_{j \in J(k)} x_j = 1,$$

where $J(k) \subseteq \{1, \ldots, n\}$ denotes the set of tours serviced by vehicle k. These inequalities fit into the set partitioning model: They give rise to additional rows that correspond to vehicles instead of requests (possibly introducing additional columns as well, that correspond to slack variables).

A set partitioning model is well suited for the VSP, because it allows a correct treatment of constraints and objectives that do not arise from individual components of a tour, but from a tour as a whole. Break rules, for instance, are observed by constructing only such tour-columns for the chaining SPP that correspond to tours with feasible breaks. If operation of a vehicle at night incurs additional costs, we can modify the objective accordingly. Or we can penalize "packed tours" that operate at full capacity because delays are more likely and try to produce safer schedules at some additional cost. A second advantage is that a correct tour matrix A already guarantees that all feasibility constraints are satisfied such that the selection of the best set of tours can be done in a second step on an abstract level: If the rules for feasible tours change, the cluster or tour matrix changes, but a solver for set partitioning problems will still be useful. This makes the approach particularly useful to analyze different scenarios of operation.

Our clustering and chaining approach to the VSP using set partitioning can now be stated as follows:

- *Clustering*
  - (i) *Cluster generation* to construct all possible clusters and set up the clustering SPP.
  - (ii) *Cluster selection* to solve the clustering SPP to select a best set of orders such that each request is contained in exactly one order.
- *Chaining*
  - (iii) *Tour generation* to construct a set of feasible tours and set up the chaining SPP.
  - (iv) *Tour selection* to solve the chaining SPP and thus choose a best set of tours.

The approach requires an implementation of three modules: a *cluster generator*, a *tour generator*, and a *set partitioning solver*. Our cluster generator is based on complete enumeration. It turned out that there are usually about 100,000 to 250,000 legal clusters in a typical VSP that can be produced in a couple of minutes. The corresponding set partitioning problems are of a size that can be solved to near or proven optimality using branch-and-cut algorithms and it is possible to do this in the Telebus case. The number of possible tours in the chaining problem is, however, much larger, and we can neither compute nor store all of them. We have nevertheless chosen to use the same branch-and-cut algorithm as for the clustering problems in the chaining instances, and we must thus restrict the set of considered tours to a (small) subset of, say, 50,000 possible tours that we construct in a heuristic way. It turned out that the chaining SPPs are computationally much harder than the clustering ones, and we cannot solve them to optimality. But our tour optimization still yields significant savings in operational costs of about 10% in comparison to what we can achieve with heuristic chaining methods.

Our set partitioning clustering and chaining approach is a *static variant* of the methods discussed in Cullen, Jarvis, and Ratliff [1981], that solve a sequence of dynamically generated set partitioning problems in both the clustering and the chaining phase using column generation techniques, or Desrosiers, Dumas, Ioachim, and Solomon [1991], that use dynamic programming techniques in a column generation algorithm for the clustering problem. An overview on related techniques and pointers to the extensive literature on vehicle routing problems can be found in the survey articles Desrochers, Desrosiers, and Soumis [1984] or Desrosiers, Dumas, Solomon, and Soumis [1995].

3 Cluster Generation

The aim of the cluster generation step is to enumerate all possible clusters. As was pointed out in the previous Section 2, we will ignore feasibility conditions for complete tours like breaks, depot locations, and shift lengths for the moment, i.e., we ignore all information related to vehicle groups. Different vehicle types (Teletaxi, small and large 1- and 2-bus), however, give rise to different possible clusters. We can deal with this parameter by enumerating the clusters for each of the five types separately. For ease of exposition, we can thus assume that there is only one type of vehicles that can service all requests.

A way to enumerate all possible clusters in a systematic way is to consider the operation of the vehicle in a cluster as the result of a sequence of decisions to pick-up or deliver a next customer, or, in other words, to add a next node to the cluster sequence. Each time this is done, the vehicle must drive to the corresponding node and pick-up or deliver the customer, before the next decision can be taken.

The possible *states* of the vehicle can be recorded in terms of cluster subsequences

$$S = (v^1, \ldots, v^l),$$

where each node v^j denotes a pick-up or delivery node of some request. We adopt the convention that a vehicle in state S has just serviced the last pick-up or delivery node v^l. More information on the vehicle can be derived from this basic state description. First, there is the set of yet unserviced pick-up nodes

$$R(S) := \{v_i^{\text{pick-up}} : \exists v^j = v_i^{\text{pick-up}}, \nexists v^j = v_i^{\text{target}}\}.$$

The customers of the unserviced requests are sitting in the car that has at state S a total load of

$$a(S) := \sum_{v_i^{\text{pick-up}} \in R(S)} a_i.$$

Since it is forbidden to wait with a customer on board, the total time since service of the sequence S began is independent of the precise starting time and amounts to

$$t(S) := \sum_{j=1}^{l} t(v^{j-1}, v^j) + t^{\text{service}}(v^j),$$

where $t(v^{j-1}, v^j)$ denotes the time to drive from node v^{j-1} to node v^j and where $v^0 := v^1$ such that $t(v^0, v^1) = 0$. Depending on the time intervals associated to the nodes in S, the service of the complete sequence S may be shifted back or forth over a certain feasible time interval. This results in intervals of feasible times

$$T^{\text{pick-up}}(S) \quad \text{and} \quad T^{\text{target}}(S)$$

to start service of the sequence and to end service at the currently last node v^l. Since the total service time $t(S)$ is a constant, we have that these intervals have the same length and, in fact,

$$T^{\text{pick-up}}(S) + t(S) = T^{\text{target}}(S).$$

We will discuss in a second how $T^{\text{pick-up}}(S)$ and $T^{\text{target}}(S)$ can be computed iteratively.

With this terminology, we can devise a simple algorithm to enumerate all possible clusters. We start by setting S to an *initial state*

$$S := (v_i^{\text{pick-up}}).$$

Then,

$R(S) = \{v_i^{\text{pick-up}}\}$	request i is not yet serviced
$a(S) = a_i$	customers and companions of request i are in the car
$t(S) = t^{\text{service}}(v_i^{\text{pick-up}})$	the total time spent to service the cluster was used to pick-up request i
$T^{\text{pick-up}}(S) = T(v_i^{\text{pick-up}})$	service of the cluster can start whenever i is eligible for pick-up
$T^{\text{target}}(S) = T(v_i^{\text{pick-up}}) + t(v_i^{\text{pick-up}})$	service of the cluster ends in the same interval shifted back the serviced time $t(v_i^{\text{pick-up}})$

We can now decide for the next node to service and this decision will lead to a transition to a new state. In general, a *state transition* from a state S to a state S' servicing an additional node v^{l+1} servicing request i looks as follows:

$S' := (v^1, \ldots, v^{l+1})$	the new node v^{l+1} is added to the cluster subsequence
$R(S') = \begin{cases} R(S) \cup \{v^{l+1}\} & \text{if } v^{l+1} = v_i^{\text{pick-up}} \\ R(S) \setminus \{v^{l+1}\} & \text{if } v^{l+1} = v_i^{\text{target}} \end{cases}$	a request is serviced or there is another customer to be serviced
$a(S') = \begin{cases} a(S) + a_i & \text{if } v^{l+1} = v_i^{\text{pick-up}} \\ a(S) - a_i & \text{if } v^{l+1} = v_i^{\text{target}} \end{cases}$	customers and companions of request i enter/leave the car
$t(S') = t(S) + t(v^l, v^{l+1}) + t^{\text{service}}(v^{l+1})$	total time to service the cluster goes up by time to drive from v^l to v^{l+1} and to service v^{l+1}
$T^{\text{target}}(S') = ((T(S) + t(v^l, v^{l+1})) \cap T(v^{l+1})) + t^{\text{service}}(v^{l+1})$	possible times to complete service of v^{l+1} are as follows: service at v^l ends in $T(S)$, the vehicle arrives at v^{l+1} in $T(S) + t(v^l, v^{l+1})$, but feasible times are in $T(v^{l+1})$, time $t^{\text{service}}(v^{l+1})$ passes until the request is serviced
$T^{\text{pick-up}}(S') = T^{\text{target}}(S') - t(S')$	the time interval to start service of the cluster is possibly reduced

We will denote this state transition by

$$S' := S \leftarrow v^{l+1}.$$

Not all states that we can produce in this way are feasible or correspond to a cluster. Conditions for a feasible state S for some vehicle k are

$a(S) < A_k$	the load does not exceed the vehicle's capacity
$T^{\text{pick-up}}(S) \neq \emptyset$	all customers can be picked up in time
$T^{\text{target}}(S) \neq \emptyset$	all customers can be delivered in time

Other feasibility conditions are that a state S must contain a node only once and that each target node must be preceded by the corresponding pick-up node and vice versa. A state that does not satisfy all of these conditions is called *infeasible*. The state corresponds to a cluster c when $R(S)$ becomes empty; such a state is called *terminal*. In this case, we can set

$$\begin{aligned} &S_c := S \\ &T_c^{\text{pick-up}} := T^{\text{pick-up}}(S) \\ &T_c^{\text{target}} := T^{\text{target}}(S) \\ &t_c := t(S). \end{aligned}$$

(The vehicle type was fixed at the beginning of this Section by assumption.)

```
void dfs (state S, digraph D)
{
    if (infeasible (S) || eliminated (S)) return;
    if (terminated (S)) output (S);

    // service next request
    for all feasible transitions $v^{k+1} \in \gamma^+(v^k)$
        dfs (S $\leftarrow v^{l+1}$, D);
}

void cluster (digraph D=(V,A))
{
    for all pick-up nodes $v_i^{\text{pick-up}}$ $\in$ V
        dfs (initial ($v_i^{\text{pick-up}}$), D);
}
```

Figure 8: Generic Cluster Generation.

A simple algorithm to enumerate all possible clusters is to consider all possible initial states and, starting from these, to recursively do all possible state transitions. The recursion stops when a terminal or infeasible state is reached, the terminal states are returned.

Most state transitions, however, will immediately lead to infeasible states, and some effort must be spent to filter these out. We do this using a *transition digraph* $D = (V, A)$, whose vertices are the pick-up and target nodes. There is an edge from node u to v if

$$(T(u) + t^{\text{service}}(u) + t(u, v)) \cap T(v) \neq \emptyset,$$

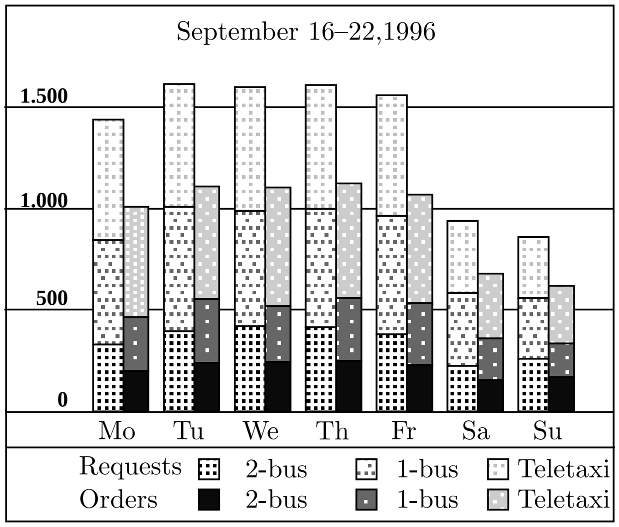
that is, if there is some feasible time to arrive at u, service u, drive to v, and arrive there at a feasible time. Since the target time interval $T^{\text{target}}(S)$ of some state S with terminal node v^l is always a subset of $T(v^l) + t^{\text{service}}(v^l)$, only the heads $\delta^+(v^l)$ of the arcs that go out from v^l qualify as candidates for feasible transitions.

Other states that must turn infeasible contain unserviced pick-up nodes $v_i^{\text{pick-up}}$ such that the corresponding target nodes can no longer be reached in time. An easy criterion to detect this is

$$\max T(v_i^{\text{target}}) < \min T(S) + t(v^l, v_i^{\text{target}}),$$

that is, when it is impossible to arrive at the target node of the unserviced request i in time even if we go there immediately. One can work out more elaborate *state elimination criteria*, but for Telebus, this one proved to be efficient enough.

C-type pseudocode for our generic recursive procedure to enumerate all clusters (for a fixed vehicle type) is given in Figure 8. The procedure searches in a depth-first way starting from all possible initial states. `digraph` is a data structure to store the transition digraph, and `D=(V,A)` is this digraph as produced somewhere else. `state` is a data structure for cluster subsequences that contains the data items discussed in this Section. `infeasible`, `eliminated`, and `terminal` are boolean functions that check a state for infeasibility, whether it can be eliminated, or is terminal as described above. `initial` is a function that returns an initial state corresponding to a pick-up node, `output` saves the cluster associated to a terminal subsequence to some medium.



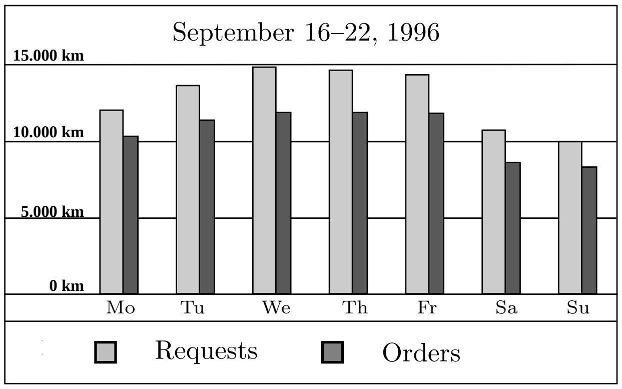


Figure 9: Clustering Requests.

Our procedure for cluster enumeration at Telebus is very simple: We do not use a dynamic program, and our state space elimination criteria are straightforward. There are two reasons why this algorithm is successful for the Telebus instances. One is the ratio of service time, transportation time, and maximum detour time at Telebus. Service of a request takes about 30 minutes on average: 5 minutes pick-up service, 20 minutes driving, and another 5 minutes of service at the target. Since a customer is not satisfied if his transportation takes more than, say, 15 minutes longer to pick-up or drop somebody else, it is often just not possible to service more than two requests simultaneously. A second reason is that BZA rules do not accept all clusters as produced by the above generic cluster generation routine. In fact, there is a catalogue of "legal" clusters at Telebus, consisting of collections, insertions, concatenations, and continued concatenations of a maximum "depth" (currently at most 3). We use more restrictive derivatives of the generic routine to produce the legal clusters and these are, of course, less than what the generic routine would yield.

The cluster generators routines usually produce, depending on the requests, the complete set of 100,000 to 250,000 legal clusters in a couple of minutes. The resulting set partitioning problems are large-scale, but computationally not difficult in the sense that one can find near or proven optimal solution in about the same time. Optimizing the internal travelling distance of the vehicles within the clusters, one obtains a reduction of about 20% in comparison to individual transportation, while the number of orders is up to 40% less than the number of requests. Figure 9 gives a graphical impression of these reductions.

4 Tour Generation

The aim of the tour generation step is to produce feasible vehicle tours as sequences of clusters. The basic flavour is similar to cluster generation where service nodes are replaced by complete clusters. But where clustering had an eye on local optimization and ignored *tour feasibility* conditions, vehicle group information like depot locations, break rules, and shift lengths must be considered in tour construction. Another difference is that while it is not allowed to interrupt the service of cluster, it is not only legal, but often advantageous to wait between service of two clusters.

We deal with different *vehicle groups* by constructing the tours for vehicles of each group separately and will assume in the remainder of this section that we have fixed a depot location, the shift length, and the vehicle type similar to what we did in cluster generation. We can then also assume that all clusters can be serviced by the vehicles of the group under consideration.

Again analogous to cluster generation, our approach to chaining is to iteratively build tours as sequences of clusters, but with an additional eye on tour feasibility criteria , and represent the possible *states* of a vehicle in terms of a tour subsequence of serviced clusters

$$S = (c^1, \ldots, c^l);$$

the interpretation of state S is that the vehicle has just completed service of the terminal cluster c^l.

The main difference between clustering and chaining is the additional consideration of driver breaks and shift lengths. Both criteria are in terms of *total elapsed time* since the start of the tour: The shift length simple sets an upper bound to this value, the break rule prescribes an obligatory break of 30 minutes between the fourth and sixth hour of work. Our approach to get control over the total time is simply to consider all possible times when a tour can start separately. All possible times means in this case every quarter of an hour, because 15 minutes is the minimum accounting unit of the vehicle providers.

We can model the different possibilities of *pull-in* times t^{pullin} to start a tour by means of a "pull-in" cluster c^{pullin} with

$S_{c^{\text{pullin}}} = (v^{\text{pick-up}}_{\text{pullin}}, v^{\text{target}}_{\text{pullin}})$	pull-in cluster (starts and) ends at the depot location
$T^{\text{pick-up}}_{c^{\text{pullin}}} = [t^{\text{pullin}}, t^{\text{pullin}}]$	pull-in time of tour
$T^{\text{target}}_{c^{\text{pullin}}} = T^{\text{pick-up}}$	pull-in time of tour,
$t_{c^{\text{pullin}}} = 0$	no service

that represents the start of a vehicle tour and will be used to initialize the cluster sequence of the tour. The pull-in cluster contains two service nodes with service time zero, that point to the depot location. There is a unique feasible pick-up time such that the pull-in cluster fixes the starting time of a tour. An analogous *pull-out* cluster is supposed to terminate the tour. Its service time intervals are chosen to model the shift length, i.e., for an $8\frac{1}{2}$ hour shift we would have

$$T^{\text{pick-up}}_{c^{\text{pullout}}} = T^{\text{target}}_{c^{\text{pullout}}} = T^{\text{pick-up}}_{c^{\text{pullin}}} + 8.5 * 12$$

(an hour contains 12 units of 5 minutes).

When the starting time t^{pullin} of the tour is fixed, breaks can be modelled by a *break cluster* c^{break} with

$S_c = \emptyset$	no pick-up and target node
$T^{\text{pick-up}}_{c^{\text{break}}} = t^{\text{pullin}} + [4, 5.5] * 12$	feasible time interval to start break
$T^{\text{target}}_{c^{\text{break}}} = t^{\text{pullin}} + [5.5, 6] * 12$	feasible time interval to end break
$t_{c^{\text{break}}} = 6$	duration of break

($6 * 5 = 30$ minutes). that has to be serviced by the tour. We adopt here the convention that an empty cluster sequence results in a standstill of the vehicle at its current location. Our goal is to construct all cluster sequences that start at a fixed pull-in cluster, contain the fitting break cluster, and end at the corresponding pull-out cluster.

An algorithm for this must derive and update only a single data item from a state S, the interval

$$T^{\text{target}}(S)$$

of feasible times to end service of the last cluster in the tour subsequence, and even here only the earliest such time $\underline{t}^{\text{target}}(S)$ is relevant, because one can always wait arbitrarily long to service the next cluster.

The algorithm starts in a (fixed) *initial pull-in state*

$$S = (c^{\text{pullin}})$$

with

$$T^{\text{target}}(S) = T^{\text{target}}_{c^{\text{pullin}}} = [t^{\text{pullin}}, t^{\text{pullin}}].$$

We can now decide for a next cluster to service, add this to the tour cluster subsequence, and so on. In general, we will be in a state $S = (c^1, \ldots, c^l)$ and decide to service a next cluster c^{l+1}. This results in a *state transition* to the new state S' with

$S'_c = (c^1, \ldots, c^{l+1})$	c^{l+1} is the new terminal cluster
$T^{\text{target}}(S') = (T^{\text{target}}(S) + [t(c^l, c^{l+1}, \infty]) \cap T^{\text{pick-up}}_{c^l} + t_{c^l}$	feasible times to end service of cluster c^{l+1} are as follows: service of cluster c^l ends in $T^{\text{target}}_{c^l}$, $t(c^l, c^{l+1})$ is needed to drive from c^l to c^{l+1}, one possible waits, feasible times to start service of C^{l+1} are $T^{\text{pick-up}}_{c^{l+1}}$, it takes another $t_{c^{l+1}}$ to service c^{l+1},

where $t(c^l, c^{l+1})$ is the time needed to drive from the terminal node of c^l to the initial node of c^{l+1}. We denote this state transition by

$$S' = S \leftarrow c^{l+1}.$$

Feasibility conditions for a state are

$$T^{\text{target}}(S) \neq \emptyset$$

and that each cluster is contained only once. A feasible state that contains the pull-in cluster under consideration as the initial cluster, the corresponding break cluster c^{break}, and the terminal cluster c^{pullout} is called *terminal*.

The aim of tour generation is in this terminology to enumerate all terminal states. A simple algorithm to do this is to consider all possible initial pull-in states, to recursively examine all feasible state transitions, and to output all encountered terminal states.

To make this approach work we want to consider only transitions that do not immediately lead to infeasible states because of incompatible service times. A necessary condition for the existence of a feasible transition from some cluster u to another cluster v is

$$(T_u^{\text{target}} + [t(u,v), \infty)) \cap T_v^{\text{pick-up}} \neq \emptyset,$$

i.e., it is possible to service u, drive to the initial node of v, possibly wait, and start service of v at a feasible time. We can store this set of possible follow-on clusters in another *transition digraph* $D = (V, A)$ that has an arc from cluster u to v if this condition holds. Then, $\gamma^+(u)$ is the set of possible follow-on clusters for a cluster u. But different from the situation in cluster generation, the number of possible follow-ons is very large: An hour in the future every cluster is eligible!

Elimination criteria for states that cannot lead to a terminal state focus on the break and pull-out cluster. If it is no longer possible to make a feasible break because

$$\min T^{\text{target}}(S) > t^{\text{pullin}} + 6 * 12$$

or pull-out is no longer possible because

$$\min T^{\text{target}}(S) + t(c^l, c^{\text{pullout}}) > \overline{t}_{c^{\text{pullout}}},$$

we can forget about state S.

The generic program for tour enumeration that results from these consideration is so similar to the cluster generation routine that we refrain from giving more detailed pseudocode here.

As we have already pointed out, the combinatorial situation for tour generation differs from the clustering scenario because the number of possible follow-on clusters is much higher. In fact it is not possible to produce all possible vehicle tours in this way, and the reason is not that the routine wouldn't work fast enough, but that the output is simply so big that there is no hope of even storing it. Also, the majority of tours obviously consists of rather inefficient tours, such that an optimal plan will contain only a few of them — which does of course not release us from trying to find "the right ones".

Since our set partitioning solver is a branch-and-cut code, we decided to reduce the solution space by producing only a "promising" set of tours that hopefully combine to a good vehicle schedule. Our tour generation routines are modifications of the above generic procedure that produce tours along *heuristic* strategies that we have developed in cooperation with the BZA. All of these heuristics work very fast and together they can also be used as a stand-alone vehicle scheduling module (in fact, this was a first stage of installation of the Telebus-computersystem at the BZA).

The *x best neighbors* heuristic tries to produce "good" tours by applying the generic enumeration algorithm to a restricted transition digraph where the outdegree of each cluster, i.e., the number of follow-on clusters, has been limited to some value (we use $x = 2$ and $x = 3$). The x surviving neighbors of each cluster are chosen with respect to local criteria, like "nearest clusters".

The *tour-by-tour greedy* heuristic tries to work in a slightly more global way by iteratively producing a feasible tour. It selects an initial pull-in state and adds "best fitting" clusters (including the break) until the pull-out state is reached. The serviced clusters are removed from the transition digraph, the next tour is started, and so on. This heuristic tends to produce "good" tours at the beginning and yields unsatisfactory results at the end when only far-out or otherwise unattractive clusters are left. Tour-by-tour produces complete vehicle schedules.

Time-sweep also constructs a complete schedule by scanning the clusters in some order. In every step, the next cluster is assigned to a best fitting tour (that is eventually created), until all clusters are scheduled. We use the natural orderings in time (from morning to evening and from evening to morning), and a "peaks-first" variant, that tries to smooth out peaks of demand and link the resulting subtours.

A *hybrid* time-sweep greedy heuristic performs a time-sweep, but always adds not only one, but some x best neighbors to a tour.

Of a similar flavor is the *assignment* heuristic, that subdivides the time interval into slots of half an hour, and constructs an assignment of the subtours (possibly starting new ones) to the follow-on clusters of the next slot.

A set of other methods imitates the *hand-planning* methods that were in use at the BZA earlier. These methods partition the requests by hours and city districts. Doing a time sweep from morning to evening, one looks at densities of requests in districts and hours and tries to concentrate vehicles in or near regions of high demand.

These methods can produce vehicle schedules that are already significantly superior to a comparable hand planning. We use them in this way and to set up chaining set partitioning problems with up to 100,000 columns. These IPs turned out to be computationally much harder than the clustering instances. A possible explanations is that clusters have a local nature and do not interact much, while tours extend over much larger time periods and service areas and thus exert more influence on each other. So we cannot solve the chaining set partitioning problems to optimality, but we nevertheless obtain significant reductions in operational costs of about 10% in comparison to what we can achieve by only using the chaining heuristics. There is, of course, even more potential for cost reductions if a better column generation method would be used.

5 Set Partitioning

The third module of our vehicle scheduling system for Telebus consists of a branch-and-cut algorithm to solve large-scale set partitioning problems. High-level pseudocode for the algorithm is shown in Figure 10. We will now quickly state our branch-and-cut terminology and discuss then some aspects of our implementation.

The algorithm uses a *branch-and-bound enumeration scheme* for solving set partitioning problems that is based on considering a *subproblems*

$$(\mathrm{SPP}(l,u)) \qquad \begin{array}{rcl} \min c^T x & & \\ Ax & = & \mathbb{1} \\ -x & \leq & l \\ x & \leq & u \\ x & \in & \{0,1\}^n, \end{array}$$

of the original problem, where the lower and upper bounds l and u are 0/1-vectors. The original problem reads in this notation $\mathrm{SPP}(0,\mathbb{1})$, and a subproblem is formed by setting some of the upper bounds to zero, such that the corresponding variables are fixed to zero, and some of the lower bounds to one, that is, fixings of variables to one.

The scheme computes for each subproblem $\mathrm{SPP}(l,u)$ a *lower* and an *upper bound*

$$\underline{z}(l,u) \leq z^\star(l,u) \leq \overline{z}(l,u) = c^T\overline{x}(l,u)$$

on the *optimal objective value* $z^\star(l,u)$: The lower bound is derived from the LP-relaxation $\mathrm{QSPP}(l,u)$, the upper bound and a corresponding feasible solution $\overline{x}(l,u)$ are computed by a heuristic to be discussed later; when the heuristic fails, we have $\overline{z}(l,u) = +\infty$ and $\overline{x}(l,u)$ is "undefined".

Subproblems are useful to search the solution space of $\mathrm{SPP}(0,1)$ in a divide-and-conquer way. The technique involves a rooted binary *searchtree T*, whose nodes are subproblems $\mathrm{SPP}(l,u)$. The tree is *initialized* to consists only of the root node $\mathrm{SPP}(0,\mathbb{1})$ and by setting $\underline{z}(0,\mathbb{1}) := -\infty$ and $\overline{z}(0,\mathbb{1}) := +\infty$, i.e., no lower and upper bounds for $\mathrm{SPP}(0,\mathbb{1})$ are know in the beginning. The algorithm *works* the root node by improving $\underline{z}(0,\mathbb{1})$ and $\overline{z}(0,\mathbb{1})$ and *labels* the root as being processed. If this step results in $\underline{z}(0,\mathbb{1}) = \overline{z}(0,\mathbb{1})$, the problem is solved and $\overline{x}(0,\mathbb{1})$ is the optimal solution. Otherwise, a *branching step* is taken to subdivide the problem into two subproblems $\mathrm{SPP}(l_1,u_1)$ and $\mathrm{SPP}(l_2,u_2)$, that become the sons of the root node. The subdivision must be done in such a way that the optimal solution for the root problem is contained in one of the two subproblems, in formulas:

$$\min\{z^\star(l_1,u_1), z^\star(l_2,u_2)\} = z^\star(0,\mathbb{1}).$$

Since the subproblems are restrictions of the father problem, their lower bounds are at least as large and we can initialize them

$$\underline{z}(l_1,u_1) := \underline{z}(l_2,u_2) := \underline{z}(0,\mathbb{1})$$

with the father's lower bound. In general, the algorithm picks an unlabelled node v, works, and labels it. Either the node can be solved, or a branching step is taken adding two new unlabelled subproblems as the sons of v to the tree. To guarantee finiteness of this process, the branching process is done in such a way that each subproblem has at least one stricter bound than its father. This results in one more variable fixed, and after a finite number of steps all variables are fixed and the subproblem is trivially solved.

To save work, the algorithm maintains a *global upper bound*

$$\overline{z}(T) = \max_{\text{SPP}(l,u) \text{ unlabelled node of } T} \overline{z}(l,u),$$

which is the value of the best solution encountered in any of T's subproblems. The bound can be used to *fathom* subproblems that cannot contain a better solution than the currently best know because

$$\underline{z}(l,u) \geq \overline{z}(T);$$

such nodes can be labelled immediately and will then not be considered any further.

This standard branch-and-bound algorithm leaves a lot of freedom to implement its generic subroutines. We will explain some aspects of our algorithm in the following subsections.

```
// initialization
read problem;
initial preprocessing;
set up searchtree;

// branch-and-bound loop
while (∃ unlabelled subproblem) {
    select and label unlabelled subproblem;

    // LP-plunging heuristic
    set-up local LP-relaxation;
    do {
        solve LP-relaxation;
        if (integral) {
            update $\overline{z}(T)$;
            break;
        }
        set some fractional variables to integer values;
        out-pivoting;
        preprocessing;
        in-pivoting;
    }
    while (!infeasible);

    // cutting-plane loop
    set-up local LP-relaxation;
    do {
        solve LP-relaxation;
        if (integral) {
            update $\overline{z}(T)$;
            break;
        }
        if (fathomed) break;
        out-pivoting;
        preprocessing;
        in-pivoting;
        separation;
        LP-management;
    }
    while (progress);
    branch;
}
output $\overline{z}(T)$;
```

Figure 10: Branch-and-cut algorithm.

5.1 Searchtree

The generic branch-and-bound algorithm does not specify the rule to choose the next unlabelled node. We use the so-called *best-first* rule, that chooses the node with the smallest lower bound, i.e., the node that has most potential for possible improvement of the global upper bound. The smallest lower bound is also called the *global lower bound*

$$\underline{z}(T) = \min_{\text{SPP}(l,u) \text{ unlabelled node in } T} \underline{z}(l,u).$$

The best-first choice potentially raises the global lower bound and thus decreases the *duality gap*

$$\overline{z}(T) - \underline{z}(T),$$

which is a measure of the global progress of the algorithm.

Best-first requires that we can jump from one problem in the searchtree to any other. Our implementation uses a *local setup procedure* to do this, that simply generates the complete LP-relaxation of a subproblem from scratch. This looks like a time consuming operation at first sight, but the method has advantages when additional cutting planes are used and redundant parts of the problem are removed by preprocessing techniques: Redundant parts for one subproblem are not necessarily redundant for others such that removed parts have to be restored, and similar actions are necessary if different sets of cutting planes are used in the subproblems. Removing and reinserting parts of a subproblem's description, however, takes about the same time as a set-up form scratch.

The method to derive lower bounds $\underline{z}(l,u)$ for the subproblems of the branch-and-bound tree is to solve the LP-relaxation

$$(\mathrm{QSPP}(l,u)) \qquad \begin{array}{rcl} \multicolumn{3}{l}{\min c^T x} \\ Ax & = & \mathbb{1} \\ -x & \leq & l \\ x & \leq & u \end{array}$$

of the integer program (SPP) and a crucial point is that this has not to be done from scratch every time. Rather, the *dual simplex method* allows to use the optimal solution of the father's LP-relaxation as a *dual feasible starting basis* for the LP-relaxations of its sons and often only a few iterations are needed to recover primal feasibility and thus optimality. To benefit from this favorable behavior, we store these optimal basis for later use as starting basis.

A last point to specify is the *branching rule* that we use to subdivide a subproblem into two smaller problems. We mainly use Ryan and Foster [1981]'s rule and *strong branching*, see CPLEX [1995], that perform on our instances in a similar way.

5.2 Cutting Planes and LP-Management

The LP-relaxations of the subproblems can be strengthened by adding various types of globally valid cutting planes, see, e.g., Balas and Padberg [1976]. We use *clique inequalities* and *simultaneously lifted odd-cycle inequalities* of the associated set packing polytope, see Padberg [1973], and a class of set covering inequalities that arise from an associated set packing problem via "complementing" and "aggregating" variables, see Borndörfer [1997]. Clique inequalities are separated both heuristically and by an exact branch-and-bound algorithm, cycle inequalities are separated using the exact polynomial algorithm of Grötschel, Lovász, and Schrijver [1988] and a Chvátal-Gomoroy simultaneous lifting procedure, and the covering inequalities by heuristic procedures. Details of these methods are discussed in Borndörfer [1997].

Working on a subproblem means to iteratively solve and strengthen the LP-relaxation by adding violated cutting planes until the subproblem is either solved, fathomed, or some other stopping criterion is satisfied and we branch. In our implementation, we use the duality gap

$$\overline{z}(l,u) - \underline{z}(l,u)$$

as a measure of progress of the cutting plane loop and continue as long as this gap is reduced by 10% in every three successive iterations.

We also remove rows from a subproblem's LP-relaxation, because the time to solve LP-relaxations of set partitioning problems increases with the number of rows of the constraints matrix. Another important point in a branch-and-cut framework is that more rows also tend to produce more fractional variables in the LP solution. To reduce running time and get a more integral solution, it is thus important to remove redundant cutting planes from a subproblem's description and we do this heuristically when the slack exceeds 10^{-3}. Each subproblem involves thus a different subset of all cutting planes that we have found somewhere throughout the course of the algorithm and if we want to be able to reproduce a subproblem exactly in the local set-up step, we must maintain a global *pool* of all cutting planes. An advantage of this method is that the computation on invocation of a subproblem becomes independent of the history of the branch-and-bound algorithm.

The LPs themselves are solved using the CPLEX dual steepest edge simplex algorithm, see CPLEX [1995].

5.3 Problem Reduction and Pivoting

Significant speed-ups for the solution of the LP-relaxations of the subproblems can be achieved by removing redundant parts like columns of variables that are fixed to zero or one, or rows that intersect columns that are fixed to one. Such fixings do not only arise from branching decisions, but also from the logical structure of a set partitioning problem, and *preprocessing* is the use of simple techniques to detect such redundancies. Preprocessing techniques for set partitioning problems are know to be highly effective, and our code uses a concept of *repeated problem reduction* that applies preprocessing techniques after each individual LP-solution. Repeated preprocessing of a similar type has been used by Atamturk, Nemhauser, and Savelsbergh [1995] for a Lagrangian heuristic for SPPs, but the technique does not seem to have been tried in a branch-and-cut framework before.

The *preprocessing techniques* that we use include know ones from the literature, like elimination of duplicate columns and rows, fixing of singletons, elimination of columns that are neighbors of a variable fixed to one, dominated rows, and some new ones. These procedures must be applied several times, because elimination of dominated rows can lead to more duplicate columns, etc. Our preprocessor performs another pass as long as it detects redundancies.

An important point in a dual simplex framework is the proper linking of preprocessing and LP-solving: Preprocessing must not destroy dual feasibility of the basis, because otherwise we would have to solve the LP essentially from scratch. The consequence is that we are not allowed to remove fixed *basic* variables and we cannot remove redundant *nonbasic* rows. The desire to remove such redundant parts of the problem nevertheless leads to some algorithmic consequences that we explain now.

Dual feasibility of the basis forces us to distinguish between *fixings* and *eliminations* of variables. Fixing is the setting of bounds of variables, elimination involves a real removal of data from memory. Our preprocessor works only with fixings, a subsequent elimination removes all fixed non-basic variables and all detected redundant basic rows from memory. In this way, we combine a maximum of problem reduction with maintenance of the basis's dual feasibility. Nevertheless, one would like to remove all detected redundancies from memory, and this leads to the consideration of *pivoting techniques*. The aim of these techniques is simply to perform a number of (degenerate) pivots to move from one optimal basis to an alternative one, such that all fixed variables are nonbasic, all detected redundant rows are basic (have their slack/artificial variable in the basis), and all of these redundancies can be eliminated. A pivoting technique that is implemented in CPLEX is the *in-pivoting* of rows with zero dual multiplier into the basis. It unfortunately turns out that most of the (known) redundant rows have nonzero duals and the reason is that fixed variables tend to proliferate in the basis: Often more than 30% of the basis consists of "junk" of this type, inhibiting removal of the same number of rows. Fixed variables can also be pivoted out of the basis using dual simplex steps, and we are grateful to Robert E. Bixby that we have access to a version of CPLEX that provides this novel out-pivoting routine. Application of the procedure usually leads to a faster problem reduction, but out-pivoting is not cheap: It requires one dual pivot for each fixed variable. One thus has to compare the benefits of eliminating large numbers of fixed variables by a consequently large number of pivots with the possibly few simplex iterations required to solve the next LP without prior pivoting. Eliminations, however, are inherited by all offspring problems and our computational experience is that out-pivoting is worth its price.

5.4 Primal Heuristics

We use the popular *LP-plunging* heuristic to generate upper bounds and feasible solutions for a subproblem in the searchtree. This heuristic solves the LP-relaxation of a subproblem, fixes some fractional variables to integer values, and iterates, until the solution becomes integral or the problem infeasible. Our algorithm does not have a separate implementation of this routine, but simply uses the main cutting-plane loop in a "primal mode" where separation is turned off. This results in particular in iterative preprocessing after each fixing decision, and this results in a fast reduction of the problem size. The heuristic is nevertheless expensive: A sequence of LPs has to be solved, and the elimination of (the largest) parts of the associated data forces a subsequent second local setup of the subproblem to initiate the cutting-plane loop. We thus call the heuristic only once at the invocation of a new subproblem.

6 Computational Results

In this section we report on computational experiences with our vehicle scheduling system. Our aim is to discuss two complexes of *questions*. Our first and main goal is to evaluate the usefulness of our set partitioning approach for the solution of VSPs at Telebus. Does clustering lead to savings in internal travelling distance? Does tour optimization lead to better results than our heuristics? Second, we want to look at the performance of our software modules for Telebus instances. What is the size of the problems that we can solve in reasonable time? What is the quality of the solutions?

To answer the second question, we ran our branch-and-cut algorithm on a *test set* of Telebus clustering and chaining problems. It is not interesting to provide performance data for the cluster and tour generators, because there is no computational bottleneck in these procedures. Our branch-and-cut code is implemented in `C` and consists of about 1 MB of source code in 140,000 lines, the LP-solver is the CPLEX Callable Library V4.0, CPLEX [1995]. All test runs were made on a Sun Ultra Sparc 1 Model 170E, the code was compiled with the Sun `cc` compiler using the switches `-fast -xO5`, and we used a time limit of 7,200 CPU seconds. The format of the upcoming tables is as follows. Column 1 gives the name of the problem, columns 2-4 contain the size of the problem in terms of the number of rows, columns, and nonzeros. The next three columns give the number of rows, columns, and nonzeros after the initial preprocessing of the problem at the root node. Comparing columns 2-4 to columns 5-7 shows the performance of our preprocessing. The next three columns give solution values. $\underline{z}$ reports the value of the global lower bound. This number coincides with the global upper bound $\overline{z}$, when the problems is solved to proven optimality. Otherwise, we are left with a duality gap that we report in percent of the global upper bound, i.e., the gap is computed as $(\overline{z} - \underline{z})/\overline{z}$. The following five columns gives information about the performance of the branch-and-cut algorithm. There are, from left to right, the number of in- and out-pivots (Pvts), the number of cutting planes (Cuts), the number of simplex iterations to solve the LPs (Itns), the number of LPs solved (LPs), and the number of branch-and-bound nodes (B&B). The next five columns show timings: The percentage of the total running time spent in problem reduction (PP), pivoting (Pvts), separation (Cuts), LP-solution (LPs), and the heuristic (Heu). The last column gives the total running time in CPU seconds.

Our first set of test problems consists of 14 clustering problems for the weeks of April 15–22, 1996, (v0415–v0421) and the already well-known week September 16–22, 1996, (v1616–v1622) that we used to produce most of the diagrams in this article. The first five problems of each data set correspond to the weekdays Monday to Friday, the last two show a significantly smaller number of rows (= requests) and belong to the weekend. The two test sets were generated with different parameter settings of the cluster generator.

In April 1996, rules for legal clusters were very restrictive: Continued concatenations and insertions were not allowed, maximum detour time was small, etc. The cluster generator found thus only relatively few feasible clusters and the clustering SPPs are small. Moreover, most of the legal clusters provide simultaneous service of very few requests: The average number of nonzeros per column is a little above two for four days of the week, the three larger instances have a higher average because they contain many clusters for a couple of large collective requests, but the remainder of the problem has the same characteristic. This means that individual clusters do not interact a lot, the problem sort of decomposes and becomes easy. And in fact: The initial call to the preprocessor is very successful, in particular the number of rows is reduced by more than 50%. This trend continues in the branch-and-cut phase: All problems are solved to proven optimality in at most 3 minutes, and we can see from the pivoting (Pvts) and preprocessing (PP) columns that the problem is basically solved by iterative preprocessing. In particular, the high number of out-pivots shows that variables could be fixed in large numbers and the sizes of the problems were reduced very fast.

In September 1996, rules were much more liberal: The clustering problems contain, for example, continued concatenations of a depth of up to 6. Consequently, there are much more possibilities for feasible clusters, the instances are larger, contain about 4 NNEs per column, and there is more overlap. This time, the initial preprocessing is still successful, but the number of rows is reduced far less than in the first test set. And in fact, the instances turn out to be harder in the sense that we cannot solve them to proven optimality as fast: In fact, there are three instances that we cannot solve completely within our time limit of 7,200 CPU seconds. Looking at the performance of the algorithm, we see that pivoting and preprocessing need most of the time, but are successful (remember that every pivot indicates a fixed variable). However, even though we find a significant number of cutting planes, the quality of the cuts does not suffice to prevent the algorithm from extensive branching, as can be seen from the B&B column. All of this effort is, however, only spent in proving the optimality of a solution of very good quality. To show this, we have run the algorithm another time with a time limit of 2 minutes, and we see that satisfactory solutions can

	Original Problem			Preprocessed			Solutions			Branch-and-Cut					Times in %					
Name	Rows	Cols	NNEs	Rows	Cols	NNEs	$\underline{z}$	$\overline{z}$	Gap	Pvts	Cuts	Itns	LPs	B&B	PP	Pvts	Cuts	LPs	Heu	Total Time
v0415	1518	7684	20668	598	4536	10988	2429415	2429415	0.000%	12774	70	755	36	9	32	32	4	12	3	5.68
v0416	1771	19020	58453	812	11225	33991	2725602	2725602	0.000%	325151	1305	4677	1970	643	19	28	4	15	3	120.53
v0417	1765	143317	531820	715	55769	206131	2611518	2611518	0.000%	61309	294	1360	171	41	35	21	8	10	3	174.07
v0418	1765	8306	20748	742	4957	11177	2845425	2845425	0.000%	12203	81	941	25	7	29	31	6	15	3	5.72
v0419	1626	15709	52867	650	7852	25052	2590326	2590326	0.000%	4106	55	801	4	1	29	17	11	17	5	3.99
v0420	958	4099	10240	417	2593	6124	1696889	1696889	0.000%	2538	47	511	4	1	28	23	8	18	5	1.31
v0421	952	1814	3119	286	1134	1437	1853951	1853951	0.000%	2304	34	317	9	3	32	18	4	18	3	0.72
v1616	1439	67441	244727	1230	52926	199724	1006460	1006460	0.000%	1295605	11123	177084	4811	1605	6	30	8	41	8	4219.41
v1617	1619	113655	432278	1409	85457	336147	1102357	1102586	0.021%	15257970	16169	67051	15661	3571	22	46	6	10	3	7200.61
v1618	1603	146715	545337	1396	90973	349947	1152989	1154458	0.127%	2418105	5549	70533	1461	296	11	27	16	21	9	7222.28
v1619	1612	105822	401097	1424	85696	336068	1156072	1156338	0.023%	5774346	9040	124824	4203	880	15	40	15	12	10	7205.74
v1620	1560	115729	444445	1365	89512	353689	1140604	1140604	0.000%	7460098	20801	111073	19230	8161	19	29	14	11	2	5526.43
v1621	938	24772	76971	807	16683	54208	825563	825563	0.000%	12214	130	1415	13	5	23	20	16	22	3	13.79
v1622	859	13773	41656	736	11059	35304	793445	793445	0.000%	13325	99	1147	14	3	27	29	10	18	3	9.69
v1616	1439	67441	244727	1230	52926	199724	1006261	1006460	0.020%	83501	828	6193	70	11	19	26	17	22	4	125.06
v1617	1619	113655	432278	1409	85457	336147	1101822	1103036	0.110%	48690	426	2972	20	4	14	22	27	20	3	137.51
v1618	1603	146715	545337	1396	90973	349947	1152150	1156417	0.369%	38494	436	2976	15	3	12	18	32	22	2	130.19
v1619	1612	105822	401097	1424	85696	336068	1155336	1157851	0.217%	48584	528	3228	15	3	13	17	35	21	3	146.06
v1620	1560	115729	444445	1365	89512	353689	1140238	1142159	0.168%	35910	377	2940	15	3	12	20	24	29	3	133.33
v1621	938	24772	76971	807	16683	54208	825563	825563	0.000%	12214	130	1415	13	5	22	20	17	22	3	13.82
v1622	859	13773	41656	736	11059	35304	793445	793445	0.000%	13325	99	1147	14	3	26	29	10	19	3	9.40
21	29615	1375763	5070937	20954	952678	3625074	31105431	31117511	0.039%	32932766	67621	583360	47774	15258	15	34	12	18	6	32405.34

	Original Problem			Preprocessed			Solutions			Branch-and-Cut					Times in %					
Name	Rows	Cols	NNEs	Rows	Cols	NNEs	$\underline{z}$	$\overline{z}$	Gap	Pvts	Cuts	Itns	LPs	B&B	PP	Pvts	Cuts	LPs	Heu	Total Time
t0415	1518	7254	48867	870	3312	20592	5163849	5590096	7.625%	1291268	2029	93675	724	167	5	11	14	17	53	7218.94
t0416	1771	9345	62703	974	3298	19692	5882041	6130217	4.048%	1334745	2163	92796	641	144	5	11	14	17	54	7207.46
t0417	1765	7894	54885	897	3774	24186	5656886	6043157	6.392%	1614510	994	51439	316	71	6	17	6	11	60	7310.58
t0418	1765	8676	66604	999	4071	29368	6185168	6550898	5.583%	629332	1066	67551	399	87	3	9	12	21	56	7239.54
t0419	1626	9362	64745	904	3287	19990	5689134	5916956	3.850%	1891101	1235	57831	429	100	6	16	9	11	59	7251.57
t0420	958	4583	27781	562	1872	10271	4036526	4276444	5.610%	3989264	4440	135766	1507	362	10	16	11	11	52	7208.44
t0421	952	4016	24214	557	1691	9015	4113080	4354411	5.542%	4238861	4581	134126	1594	375	10	16	12	11	51	7213.44
t1716	467	56865	249149	467	11952	61110	122408	161636	24.269%	1230592	886	39379	296	69	2	7	3	11	76	7212.95
t1717	551	73885	325689	551	16428	85108	135539	184692	26.613%	1021307	592	27888	183	41	2	7	2	10	77	7331.93
t1718	523	67796	305064	523	16310	83984	127040	162992	22.058%	982755	606	28048	203	44	2	6	3	10	78	7238.72
t1719	556	72520	317391	556	15846	83893	139332	187677	25.760%	992993	404	22565	169	37	2	6	2	8	80	7281.77
t1720	538	69134	310512	538	16195	84194	127225	172752	26.354%	899591	688	30360	187	38	2	6	3	11	77	7349.28
t1721	357	36039	148848	357	9043	44106	104698	127424	17.835%	1965867	1921	69853	765	174	3	9	5	12	69	7243.42
13	13347	427369	2006452	8755	107079	575509	37482926	39859352	5.962%	22082186	21605	851277	7413	1709	4	11	7	12	65	94308.04

Table 1: Clustering and chaining.

be obtained in this period.

The clustering results are satisfactory in the sense that more or less independent of the parameter settings clusterings of proven optimality or with very good quality guarantee can be computed in short time, considering the complete solution space of all legal clusters.

We have used the clusterings that we computed in the previous test runs to set up the corresponding chaining problems as well. The April instances (t0415–t0421) contain duplicate rows for clustered requests and have thus the same number of rows as the corresponding clustering instances, the optimization criterion was operational costs, in the September instances (t1716–t1721) only the bus clusters were chained, the optimization criterion was travelling distance. Chaining rules were again more strict for April and the resulting instances are not very large in terms of columns and NNEs. Looking at the preprocessed instances (with the duplicate rows removed), however, the average number of NNEs is already larger, indicating a more complicated combinatorial structure. This can also be observed for the September instances: Here, our preprocessor cannot even remove a single row in any of the instances. Although the preprocessed instances are not large, they turn out to be computationally difficult. In contrast to what is usually reported about real world set partitioning problems, there is a large duality gap between the value of the LP-relaxation and the best know integer solution. In fact, even the duality gaps on *termination* of the algorithm as reported in column Gap are significant, in the case of the September instances even big. Most of the computational effort is spent in the heuristic, because the iterative preprocessing doesn't reduce the problems a lot in the early rounding steps. But even if we subtract this time completely, the algorithm performs comparably few iterations: The number of LPs is rather small and the same holds for the size of the searchtree. The reason for this is that the LP-relaxations of the chaining problems are harder to solve, as can be seen by looking at the average number of simplex iterations per LP (column Itns divided by column LPs).

Day	Requests	Heuristics				Integer Programming		
		Clusters		Tours		Cluster		Tours
	No.	No.	km	DM	DM	No.	km	DM
Mo	1439	1167	10909	66525	60831	1011	10248	55792
Tu	1619	1266	11870	71450	67792	1106	11291	62696
We	1603	1253	12701	74851	68166	1107	11813	61119
Th	1612	1276	12697	74059	68271	1121	11821	64863
Fr	1560	1242	12630	71944	63345	1080	11757	61532
Sa	938	748	9413	45842	47736	676	8561	41638
Su	859	703	8850	42782	44486	620	8243	38803
$\sum$	9630	7655	79070	447453	420627	6721	73734	386443

Table 2: Comparing vehicle schedules.

Although the chaining step does not provide near optimal solutions, tour optimization is still valuable. Table 2 shows the results of a comparison of different vehicle scheduling methods for the week September 16–22, 1996. Column 1 gives the day of the week and column 2 the number of requests. The next three columns show the results of a heuristic vehicle scheduling using our cluster and tour generators as a stand-alone optimization module. There are, from left to right, the number of clusters obtained from a heuristic clustering, the internal travelling distance within these clusters, and the costs of a vehicle schedule based on this clustering. Skipping column 6 for the moment, we can compare these numbers with the results that we obtained using the set partitioning approach. Column 7 gives the number of clusters obtained in this way, column 8 the corresponding internal travelling distance, and the last column the costs of the vehicle schedule that was obtained by chaining the optimal set of clusters and solving the resulting chaining SPP approximately. Column 6 that we just left out gives the costs of a vehicle schedule that was constructed heuristically from the optimal clustering. Roughly speaking, these number show that an optimal clustering reduces the number of requests about 10% better than what we can achieve heuristically. Heuristic chaining based on optimized clusters results in vehicle schedules that are about 5,000 DM per day cheaper than a pure heuristic approach, while chaining optimization can save another 5,000 DM per day.

7 Summary

We have presented in this paper a set partitioning approach to vehicle scheduling in a dial-a-ride system for handicapped people. The results show that it is today possible to solve vehicle scheduling problems for systems of this size in a satisfactory way. In the Telebus case, the use of modern computer technology and mathematical programming techniques resulted in improvements in service quality and *simultaneous* significant cost reductions. We think that such results can lead to a renewed interest in dial-a-ride systems for use not only as a special purpose system for handicapped people, but as a component of the public transport to service areas or times of low demand.

References

Atamturk, A., Nemhauser, G. L., and Savelsbergh, M. W. P. (1995). A combined lagrangian, linear programming and implication heuristic for large-scale set partitioning problems. Technical Report LEC - 95-07, Georgia Institute of Technology.

Balas, E. and Padberg, M. (1976). Set partitioning: a survey. *SIAM Review*, 18:710–760.

Ball, M. O., Magnanti, T. L., Monma, C. L., and Nemhauser, G. L., editors (1995). *Network Routing*, volume 8 of *Handbooks in Operations Research and Management Science*. Elsevier Science B.V., Amsterdam.

Borndörfer, R. (1997). *Packing, Partitioning, and Covering of Sets*. PhD thesis, Technical University of Berlin. To appear in 1997.

Borndörfer, R., Grötschel, M., Herzog, W., Klostermeier, F., Konsek, W., and Küttner, C. (1996). Kürzen muß nicht Kahlschlag heißen — das Beispiel Telebus-Behindertenfahrdienst Berlin. Preprint SC 96-41, Konrad-Zuse-Zentrum für Informationstechnik Berlin. To appear in *VOP*. In German available at URL: http://www.zib.de/ZIBbib/Publications/.

CPLEX (1995). *Using the CPLEX Callable Library*. CPLEX Optimization, Inc., Suite 279, 930 Tahoe Blvd., Bldg 802, Incline Village, NV 89451, USA. Information available via WWW at URL: http://www.cplex.com/.

Cullen, F., Jarvis, J., and Ratliff, H. (1981). Set partitioning based heuristics for interactive routing. *Networks*, 11:125–143.

Desrochers, M., Desrosiers, J., and Soumis, F. (1984). Routing with time windows by column generation. *Networks*, 14:545–565.

Desrosiers, J., Dumas, Y., Ioachim, I., and Solomon, M. (1991). A request clustering algorithm in door-to-door transportation. Technical Report G-91-50, École des Hautes Études Commerciales de Montréal, Cahiers du GERAD.

Desrosiers, J., Dumas, Y., Solomon, M. M., and Soumis, F. (1995). *Time Constrained Routing and Scheduling*. In Ball, Magnanti, Monma, and Nemhauser [1995], chapter 2, pages 35–139.

Grötschel, M., Lovász, L., and Schrijver, A. (1988). *Geometric algorithms and combinatorial optimization*. Springer Verlag, Berlin.

Klostermeier, F. and Küttner, C. (1993). Kostengünstige Disposition von Telebussen. Master's thesis, Technische Universität Berlin.

Padberg, M. W. (1973). On the facial structure of set packing polyhedra. *Mathematical Programming*, 5:199–215.

Ryan, D. M. and Foster, B. A. (1981). An integer programming approach to scheduling. In Wren [1981], pages 269–280.

Wren, A., editor (1981). *Computer scheduling of public transport: Urban passenger vehicle and crew scheduling*. North-Holland Publishing Company.