

Vehicle routing and staffing for sedan service

Oktaý Günlük, Tracy Kimbrel, Laszlo Ladanyi, Baruch Schieber, Gregory B. Sorkin
IBM Research, Yorktown Heights, New York 10598

May 2, 2005

Abstract

We present the optimization component of a decision support system developed for a sedan service provider. The system assists supervisors and dispatchers in scheduling driver shifts and routing the fleet throughout the day to satisfy customer demands within tight time windows. We periodically take a snapshot of the dynamic data and formulate an integer program, which we solve to near-optimality using column generation. Although the data snapshot is stale by the time a solution is computed, we are able to solve the integer program quickly enough that the solution can be adopted after minor modifications are made by a fast local-search heuristic. The system described in this paper is currently in use and has improved the provider's productivity significantly.

1 Introduction

Improvements in technology are enabling new applications of optimization techniques. In this paper we describe an application in which vehicles are dispatched to meet customer demands within tight time windows. We present the optimization component of a decision support system developed for a provider of car service. Cheap, reliable communications and high-speed computing are the two key advances that have enabled real-time optimization in this setting. Cars are fitted with two-way data terminals allowing the dispatch center to maintain knowledge of drivers' states and positions at all times. Modestly priced compute servers allow us to solve and re-solve the scheduling and routing problem nearly optimally throughout the day as demands change. Despite highly dynamic data, our optimization tool is able to solve the scheduling and dispatching problem quickly enough to provide a timely schedule, i.e., before the demands have changed so much that the schedule is largely invalid. Combined with a mechanism that locally updates a schedule within seconds in response to single a new input, this yields a system in which the car service provider operates more efficiently than it could with its former manual scheduling system. The system described in this paper was installed in March, 2003 and is currently in use 24 hours a day. Productivity has increased significantly since its introduction.

We separate the *static* problem of constructing a schedule for a given set of demands and resources from the *dynamic* problem of maintaining such a schedule as data changes throughout the day. In the static problem we are given several vehicle/driver base locations, a set of drivers and a number of cars at each base, and a list of rides. Each driver has upper and lower limits on start

and end times and a wage. Each ride has a start time and a predicted duration. In addition, we are given predicted travel times from the end of one ride to the start of another, and from each of the bases to each of the rides and vice versa, and mileage costs for each travel segment. The problem is to construct a near-optimal collection of driver schedules — one covering as many of the rides as possible at the smallest possible driver and mileage cost. (The actual objective function is somewhat more complicated and is detailed later.) Each driver’s schedule must meet his shift limits and must begin and end at his base. The schedule must be feasible in the obvious sense that the driver should be able to pick up each passenger on time (within lateness allowances, described later) and deliver him to his destination before proceeding to the next. Of course, the schedule must use only as many cars as are available at each base.

In the dynamic problem, the data changes throughout the day. New reservations are made, existing ones change or are cancelled, and actual ride start times and durations deviate from the forecasts. When a driver is dispatched to a ride, the assignment is locked and becomes a constraint on the schedule. The objective is to maintain a near-optimal collection of driver schedules at all times as the data changes.

In this paper, we describe an optimization-based approach to solving the static problem and discuss how it is also used to solve the dynamic problem. Later sections will detail our precise mathematical model and the algorithms we use to solve it. Here we give a high-level description of our solution. We solve the problem on three different time scales: daily (day-ahead planning) or *offline*, periodically (say, every 15 minutes) throughout the scheduling day (*continual*), and on demand with 15-second response time in response to a single new input (*instantaneous*). Offline mode is used to develop a staffing plan for the next day. It is executed several times during the day as more information about the day-ahead rides is becoming available. A significant fraction of reservations are made or changed on the day of service. Thus it is necessary to reoptimize the schedule even while it is being executed. This is the function of continual mode optimization. The primary purpose of the instantaneous mode is to maintain in the a schedule that is at all times feasible and near-optimal with respect to constantly changing data. This mode is activated in response to a single input change, for example to reschedule a ride whose planned driver is running late on another ride. Note that continual and instantaneous modes run in parallel and thus must be synchronized.

Two optimization modules are used: an integer program (IP) solver (described in detail below in Sections 3 and 4) and a heuristic local improvement solver (described in detail in Section 5).

The IP solver is used in two of the modes, offline and continual. We chose an IP solver rather than a heuristic solver for two reasons. First, we conjecture that given enough computation time an IP solution would be of better quality than a heuristic solution, and second, the IP solution enables us also to quantify the quality of the solution based on the value of the dual solution. The heuristic local improvement solver is used in instantaneous mode and in the synchronization of the continual and instantaneous modes as described below.

Because the frequency of data changes is much higher than the frequency at which the IP solver used in continual mode can be invoked, the continual mode schedule may not be fully synchronized with the input data. Thus, using instantaneous mode, we maintain a *foreground* schedule that is always feasible with respect to the up-to-the-minute input. This is done while running continual mode in the *background* with a snapshot of the input data. When the background IP solver used in continual mode completes, assignments may be committed (locked) in the foreground schedule that

conflict with those in the continual mode solution. These locks are extracted from the foreground schedule and the remainder of the foreground schedule is discarded. The locks are then applied to the continual mode schedule and the heuristic solver patches any “holes” thus created.

Our problem is a variant of the general *multi-depot vehicle routing problem with time windows*. Due to the wide applicability and the economic importance of this problem it has been extensively studied in the vehicle routing literature; for a review see [19] and also [6]. Much of the research has focused on the design and empirical analysis of heuristics for the problem; for a survey of such empirical studies see [18, 8]. Bramel and Simchi-Levi [5] formulated the vehicle routing problem with time windows as an integer program and proposed to solve it in two phases: first to solve the linear-programming relaxation using column generation and then to find an integer solution using branch and bound. Our static solver also uses column generation to solve linear-programming relaxations of the problem. These relaxations are solved to obtain a lower bound on the cost, and in the fix-and-price process they are used to find an integer solution. Related problems with applications to airline fleet scheduling are considered in Barnhart *et al.* [3] and Rexing *et al.* [16].

Independently of and in parallel to our work, Krumke, Rambau and Torres [12] considered the problem of real-time vehicle dispatching with soft time windows. They developed a system for the dispatch of roadside-service vehicles to assist motorists whose cars become disabled. Interestingly, the solution is very similar to ours. They solve the static problem intermittently, using an integer programming formulation that is solved by repeatedly solving linear-programming relaxations using column generation. Naturally, there is no knowledge of future breakdowns. Thus, in contrast to our static problem, their dispatch problem consists of only current demand. This makes the problem somewhat simpler and enables Krumke *et al.* to use their integer programming solver in the foreground rather than in the background.

2 Formal problem definition

In this section we describe the *sedan service scheduling* (SSS) problem more precisely and present an integer programming formulation for it. For a similar approach for aircraft scheduling problems see [3] and for a general discussion of the approach see [4].

For a given geographical location, which in practice is a metropolitan area, an instance of the problem consists of two major parts. *Demand* data describe the collection of rides to be served and *resource* data consist of a collection of bases, each with a limited number of cars and drivers.

- Each driver has shift limits, a base pay rate, an overtime pay rate, and a number of user-defined attributes such as experience level and commercial licenses. A driver’s shift limits are given by minimum and maximum values for each of the shift start time, end time, and duration. The pay rate for the shift changes from the base rate to the overtime rate for the portion of the shift in excess of a given duration.
- Cars are grouped into several types based on user-defined attributes such as commercial operating permits and car models. Cars of the same type are considered to be indistinguishable. Each car type has an associated mileage cost.

- Each ride has an origin, a destination, a pickup time, a maximum lateness limit, and a “bump penalty.” Serving a ride late (arriving to pick up the customer after the pickup time but within the lateness limit) imposes a penalty dependent on the lateness. If a ride is not served, the bump penalty is applied. Rides also have associated service requirements (such as driver experience level and car or vehicle licenses specific to a location such as an airport) which, if not met, incur various penalties. More precisely, for every driver and ride, as well as for every car type and ride combination, there is a mismatch penalty (possibly zero).
- Finally, the input data includes the travel times and distances between any two significant locations, namely all bases and all ride origins and destinations. The travel-time data depends on the time of day, since traffic conditions vary.

The objective is to minimize total cost, which is the sum of the drivers’ pay, car mileage costs, and various penalty costs.

For simplicity, in the remainder of this section and the next, we will assume that late pickups are not allowed. Section 4.5 will detail the handling of lateness.

2.1 Hardness of The SSS problem

We note that the SSS problem is NP-hard as it generalizes the minimum set cover problem (Set Cover). In other words, given an instance of Set Cover, one can construct an instance of the SSS such that an optimal solution of the SSS instance yields an optimal solution of the Set Cover instance.

For the sake of completeness, we first define Set Cover formally: Given a finite set $S = \{1, \dots, m\}$ and a collection $C = \{C_1, \dots, C_n\}$ of its subsets, find a minimum-cardinality subcollection $\bar{C} \subseteq C$ such that each element of S is contained in at least one set in \bar{C} .

Theorem 1 *The sedan service scheduling problem is NP-hard.*

Proof: For a given instance of Set Cover, for each $r \in S$ we define a ride r , and for every $C_i \in C$ we define a driver i . There is a single base and all drivers belong to that base. Drivers are uniformly paid \$1 for the day. All travel times, including ride durations, are set to 1 minute. The start time for ride r is set to $2r$ and all driver shifts span the period $0, \dots, 2m + 1$, so that any driver can serve any subset of the rides without lateness. However, we set the service requests of rides so that driver i is a good match for ride r only if $r \in C_i$. If a driver is not a good match for a ride than there is a \$10 penalty for assigning the driver to that ride. Each ride is assigned a bump penalty of \$10. Finally, all mileage costs are set to zero.

The cost of a solution of the SSS instance consists of penalties and driver salaries. Since it is cheaper to add a driver (\$1) than to bump a ride or violate a preference (\$10), an optimal solution does not bump any rides or violate any ride preferences. In addition, an optimal solution of the SSS instance minimizes the number of drivers since every driver has unit cost. Therefore, an optimal solution of the SSS instance gives an optimal solution of the Set Cover instance. ■

2.2 Integer Programming Formulation

It is possible to formulate the SSS problem as a large integer program with a variable for every possible (feasible) schedule for every driver. In this formulation, a driver schedule corresponds to a round trip beginning and ending at the driver's base and is specified by the following:

- a driver with fixed shift start and end times (respecting his shift limits),
- a car type and a time window over which the car will be occupied, and
- a list of rides that will be served.

For a schedule to be feasible the driver's shift must be feasible and the travel time data must allow the rides to be served on time. Due to imposed lower bounds on the length of a driver shift, it may properly contain the time window for the car, implying that the driver is idle, with no car allocated to him, for part of his shift.

We divide the day into a fixed number of discrete time periods. Each period corresponds to a half open time interval. As we discuss later in our solution approach, the time periods can be arbitrarily small without significantly increasing the required computational effort.

Let s_{ij} denote the j^{th} schedule of driver i . We use the following notation:

T	is the index set of all time periods,
R	is the index set of all rides,
B	is the index set of all bases,
K	is the index set of all car types,
D	is the index set of all drivers available,
S	is the index set of all possible schedules,
S_d	is the index set of all schedules of driver d ,
$\text{base}(i)$	is the base of driver i .
$\text{time}(s_{ij})$	is the set of time periods (which we require to be consecutive) when the schedule s_{ij} requires a car,
$\text{type}(s_{ij})$	is the car type allocated for schedule s_{ij} ,
R_{ij}	is the set of rides served by schedule s_{ij} ,
c_{ij}	is the cost of schedule s_{ij} ,
p_r	is the bump penalty for not serving ride r ,
$\text{cars}(d, k)$	is the number of cars of type k available at base d .

We use binary variable x_{ij} to denote whether driver schedule s_{ij} is in the solution, and binary variable y_r to indicate whether ride j is bumped.

$$\min \sum_{r \in R} p_r y_r + \sum_{i \in D} \sum_{j \in S_i} c_{ij} x_{ij} \quad (1)$$

$$\text{s.t.} \quad \sum_{j \in S_i} x_{ij} \leq 1 \quad \forall i \in D \quad (2)$$

$$y_r + \sum_{i \in D} \sum_{\substack{j \in S_i: \\ r \in R_{ij}}} x_{ij} = 1 \quad \forall r \in R \quad (3)$$

$$\sum_{\substack{i \in D: \\ \text{base}(i)=b}} \sum_{\substack{j \in S_i: \\ t \in \text{time}(s_{ij}) \\ \text{type}(s_{ij})=k}} x_{ij} \leq \text{cars}(b, k) \quad \forall b \in B, \forall k \in K, \forall t \in T \quad (4)$$

$$y_r \in \{0, 1\} \quad \forall r \in R \quad (5)$$

$$x_{ij} \in \{0, 1\} \quad \forall i \in D, \forall j \in S_i \quad (6)$$

The driver constraints (2) ensure that each driver is assigned at most one schedule; the ride constraints (3) ensure that each ride is either bumped or covered by exactly one schedule; and the car constraints (4) ensure that for each base, car type, and time period, the number of cars in use is not more than the number of available cars. The cost of a schedule is the sum of all mileage costs, driver pay based on shift durations, bump penalties for rides that are not covered by the schedule, and ride mismatch penalties for both car type and driver attributes. In addition, if rides are allowed to be served late (as is the case in the continual mode), there is a penalty for each ride that is served late. This penalty depends on the difference between the scheduled pickup time and the requested pickup time.

Our solution approach consists of three stages. First we solve the linear programming relaxation of the problem to near-optimality using column generation. Next, we use a fix-and-price heuristic which combines variable fixing with column generation to produce a feasible solution to the integer program. Finally, we apply local search heuristics to the solution to improve its quality. This is done primarily to increase the confidence of the user in the output solution, as it prevents a situation where the solution can be easily manually. These stages are described in detail in the following sections.

3 Solving the linear program

In this section we describe how we solve the linear relaxation of the problem to near-optimality. To simplify and speed up the computation we actually solve a further relaxation by allowing rides

to be covered multiple times. Denote by LP this relaxed linear program:

$$\min \sum_{r \in R} p_r y_r + \sum_{i \in D} \sum_{j \in S_i} c_{ij} x_{ij} \quad (7)$$

$$\text{s.t.} \quad \sum_{j \in S_i} x_{ij} \leq 1 \quad \forall i \in D \quad (8)$$

$$y_r + \sum_{i \in D} \sum_{\substack{j \in S_i: \\ r \in R_{ij}}} x_{ij} \geq 1 \quad \forall r \in R \quad (9)$$

$$\sum_{\substack{i \in D: \\ \text{base}(i)=b}} \sum_{\substack{j \in S_i: \\ t \in \text{time}(s_{ij}) \\ \text{type}(s_{ij})=k}} x_{ij} \leq \text{cars}(b, k) \quad \forall b \in B, \forall k \in K, \forall t \in T \quad (10)$$

$$0 \leq y_r \leq 1 \quad \forall r \in R \quad (11)$$

$$0 \leq x_{ij} \leq 1 \quad \forall i \in D, \forall j \in S_i \quad (12)$$

Notice that LP has the same optimal value as the original linear relaxation, provided that deleting a ride from a driver schedule does not increase the cost of the driver schedule. When we generate a solution to the integer program via fix-and-price (see Section 4), we ensure that each ride is covered exactly once.

Since there are many car constraints 10 we treat these constraints as “cutting planes” and add them “on demand” as explained in Section 3.3.3.

Since there are far too many possible driver schedules to enumerate explicitly, the columns of the formulation are generated only as needed. This is a powerful technique that has been discussed, for example, in [4], and used in [7]. Thus we iteratively solve a linear program LP' which contains only a small subset of all possible columns, and then generate improving driver schedules: columns with negative reduced cost with respect to the current dual solution. The generated schedules are added to LP' and the next iteration follows. As discussed below, it is possible to generate a lowest-reduced-cost column for each driver efficiently. We could iterate this process until we solve LP to optimality. In practice, however, this may take too long, so from time to time lower bounds are computed for the optimal solution of LP and we stop the column generation process when the optimality gap is below a certain threshold. See [14] for a recent review of column generation approaches to solving linear programs. Algorithm 1 gives an overview of how we solve LP with column generation. In the remainder of the section we discuss the details. First the column generation process is discussed, then the lower bounding technique. Finally, various methods for speeding up computations are presented.

3.1 Generating improving columns

In this section we show that generating improving (negative-reduced-cost) columns is equivalent to solving a shortest path problem. See [3] for a similar observation. To do so, we first describe the anatomy of a column corresponding to a schedule as well as its cost structure:

Algorithm 1 Solving LP with Column Generation

- 1: Formulate an initial linear program (LP') with bump-ride variables only
 - 2: Generate a small number of sets of non-overlapping schedules that cover all rides
 - 3: Generate a small number of sets of overlapping schedules that cover difficult rides
 - 4: **repeat**
 - 5: Reoptimize LP'
 - 6: **for** randomly selected drivers **do**
 - 7: Using the dual solution generate a schedule for the driver
 - 8: **if** the corresponding column has a negative reduced cost **then**
 - 9: Add it to LP'
 - 10: Discount the duals for the rides in the schedule
 - 11: **end if**
 - 12: **end for**
 - 13: **if** car availability constraints are not satisfied **then**
 - 14: Add constraints as cutting planes until all satisfied
 - 15: **end if**
 - 16: Every 20 iterations compute a lower bound
 - 17: **until** optimal value of LP' and lower bound are close
-

$$\text{column}^T = [\underbrace{00001000}_{\text{the driver}}, \underbrace{0000100001000100000000}_{\text{rides in the schedule}}, \underbrace{00000111111111110000000000000000}_{\text{car type and time}}]$$

The first block is the set of driver constraints (8); exactly one driver is selected for the schedule. The second block has a 1 in the row of each covered ride. The third block has a set of consecutive 1's for the time periods during which a car is assigned to the schedule (car constraints (10)). Note that we assume that a driver uses only one car in a shift and this is why the 1's in this block are indeed consecutive.

The cost of a schedule is:

$$\text{Cost} = \text{Driver pay} + \text{Car mileage cost} + \text{Car/Ride and Driver/Ride mismatch penalties}.$$

The reduced cost of the corresponding column is the difference of the cost of the column and the inner product of the dual vector and the column. Given the structure of the columns, the reduced cost has the form:

$$\text{Reduced Cost} = \text{Cost} - \text{driver dual} - \sum(\text{ride duals}) - \sum(\text{car duals}).$$

Since driver and car duals are non-positive and ride duals are non-negative, the reduced cost can be interpreted as follows. If we adopt the schedule corresponding to the column, we pay the cost of the schedule, pay a penalty for using the driver and the car, and collect prizes for the rides covered.

Next we describe how to generate the column with the lowest reduced cost when the driver, his shift start time, and the assigned car type are fixed, and all rides must be served on time. We show that this is equivalent to solving a shortest path problem in a directed acyclic graph.

In this graph, two special nodes s and t represents the base of the driver at the beginning and end of his shift. In addition, there is a node for every ride. There is a directed arc (r_1, r_2) from ride r_1 to ride r_2 if it is possible to reach the start location of ride r_2 on time after serving ride r_1 . Similarly, there is a directed arc from node s to a ride (resp. from a ride to t) if that ride can be served after the driver leaves his base at the given shift start time (resp. the driver can get back to his base without violating his shift limits). Clearly, every s - t path in this graph corresponds to a feasible schedule.

Now consider a schedule of driver $i \in D$ that covers rides r_1, \dots, r_m and occupies a car of type $k \in K$ from base $b \in B$, from time period t_1 to t_n . Denoting the dual vector by π , the reduced cost of the schedule is:

$$\text{Reduced Cost} = \text{Driver pay (according to shift length)} \quad (13)$$

$$+ \sum_{j=1}^m \text{mileage}(r_j) \quad (14)$$

$$+ \text{mileage}(s, r_1) + \sum_{j=1}^{m-1} \text{mileage}(r_j, r_{j+1}) + \text{mileage}(r_m, t) \quad (15)$$

$$+ \sum_{j=1}^m (\text{penalty}(k, r_j) + \text{penalty}(i, r_j)) \quad (16)$$

$$- \pi_i \quad (17)$$

$$- \sum_{j=1}^m \pi_{r_j} \quad (18)$$

$$- \sum_{t=t_1}^{t_n} \pi_{c_{k,b,t}} \quad (19)$$

The various cost components of the reduced cost are assigned to the nodes and arcs in the graph such that the total cost of an s - t path is exactly the reduced cost of the corresponding schedule (with a constant offset) as follows. The mileage cost (14) of a ride, the car/ride and driver/ride penalties (16), and the ride dual (18) are assigned to the node corresponding to the ride. The mileage costs (15) between the rides and to and from the bases are assigned to the associated arcs.

Recall that we have created this graph for a fixed driver with a fixed schedule start time and car type to be used. If a particular ride is the last one on the schedule then it is easy to compute the time at which the driver gets back to the base. This has two implications. First, the interval over which a car is needed is known, and hence the sum (19) of the car duals for that time interval can be computed. Second, the length of the driver's shift is known, hence his pay (13) for the schedule can be computed. We assign these two cost items to the arc from the ride back to the base. For a fixed driver the driver dual (17) is a constant offset.

In this arc- and node-weighted graph the s - t paths are precisely the feasible schedules, and the cost of a path is the reduced cost of the corresponding schedule. Since all arcs go forward in time, the graph is acyclic, and the shortest path problem can be solved in time linear in the number of arcs.

To assign the car duals and driver pay to the arcs we had to assume that the car type and the driver’s start time are fixed. In our application, the number of car types is very small, predetermined, and independent of the problem instance. To limit the number of possible shift start times the drivers are assumed to start and end their shifts only at hour and half-hour boundaries. Thus the column generation problem can be solved to optimality for every driver by solving the shortest path problem for every car type and every start time.

3.2 Lower bound

It is important to note that it can take a long time to solve LP to optimality. In practice, we do not need an optimal solution (see, e.g., [14]). It is enough to find a near-optimal solution, i.e., a primal solution that comes close enough to the optimal cost of LP . (Recall that LP denotes the linear program with all possible schedules).

To prove that our solution is near-optimal we need to derive a lower bound on the optimal value of LP . This can easily be done by exploiting the driver constraints (8) that state that at most one schedule per driver can be contained in any final solution. More precisely, if π is a feasible dual solution to the partial formulation LP' , then

$$z(\pi) + \sum_{i \in D} \min_{j \in S_i} rc_{\pi}(s_j) \leq z^*(LP),$$

where $z(\pi)$ is the dual objective function corresponding to π , $rc_{\pi}(s)$ is the reduced cost of the variable corresponding to schedule s with respect to π , and $z^*(LP)$ is the optimal value of LP . Note that if LP' is solved to optimality then $z^*(LP') = z(\pi^*)$ where π^* is an optimal dual solution to LP' and the optimality gap of the linear program can be bounded as

$$z^*(LP') - z^*(LP) \leq \sum_{i \in D} \min_{j \in S_i} rc_{\pi}(s_j).$$

3.3 Improving computational performance

In the remainder of this section we describe several methods to speed up computation. These methods are independent of each other, and each of them has a positive effect whether applied individually or in combination with others.

3.3.1 Using the volume algorithm instead of the simplex algorithm

It is well known that in practice simplex-based column generation has poor convergence due to the tailing-off effect and failure to produce good dual solutions needed to compute the lower bounds; see [10] for an early reference and [13, 15] for some discussion. In some applications, so-called *stabilized column generation* techniques are used to achieve better convergence results with the simplex algorithm. See [14] for a discussion.

Instead, we use the volume algorithm to improve convergence. The volume algorithm is a subgradient-based algorithm that produces approximate primal as well as dual solutions to linear programming

problems. While we do not use the primal solutions, we have still opted for the volume algorithm instead of the subgradient algorithm since we believe it has a better termination criterion. Ordinary subgradient algorithms have only the series of converging dual solutions to decide when to terminate, while the volume algorithm has the series of converging primal solutions as well—hence better termination criteria can be devised. Our termination criterion is that the maximum primal violation must be small (at most 2%) and the relative gap between the primal and dual objectives must be small as well (0.75%).

It has been noted that, especially for large combinatorial problems, the volume algorithm is fast (much faster than the simplex method) and stable [1, 2]. In our application, due to the relatively small size of the formulation, the volume algorithm is not noticeably faster. However, the dual values provided by the algorithm are of “high quality” in the sense that it requires significantly fewer column generation iterations to obtain a formulation that has a value close to the lower bound. In our computational experience, using the volume algorithm instead of the simplex algorithm decreased the number of column generation iterations and the number of schedules generated by about 20%. In addition, with the volume algorithm, re-optimizing the LP after adding columns took approximately 10% less time on the average.

It is possible to use the dual solution $\bar{\pi}$ produced by the volume algorithm for computing lower bounds on $z^*(LP)$ since $\bar{\pi}$ is a feasible dual solution. But since $\bar{\pi}$ is not necessarily optimal, one can not be certain that $z^*(LP')$ is close to $z^*(LP)$ when $z(\bar{\pi})$ is close to the lower bound on $z^*(LP)$. To overcome this problem, we stop using the volume algorithm when $z(\bar{\pi})$ is close to the lower bound (within 1%) and start using the simplex algorithm. We terminate the simplex based column generation when we prove that $z^*(LP')$ is close to $z^*(LP)$ (within 1%). In our application, this last step typically does not require any column generation since $z(\bar{\pi})$ is very close to $z^*(LP')$.

3.3.2 Adjusting dual values

In this section we describe a heuristic that in our experience significantly decreases the number of major iterations needed in Algorithm 1. We are not aware of any similar heuristic in the literature. In our earlier experiments, we observed that we were generating many schedules that included the same few rides with the greatest prizes (dual values). To obtain a well-balanced set of columns, we discount the ride duals during the column generation phase. That is, after generating a schedule for a driver, we multiply the duals of the rides in the generated schedule by a discount factor $\alpha < 1$. In addition, we start every iteration by randomly ordering the drivers, and we generate schedules only for the first m . The values of α and m change as the algorithm progresses. At the beginning of the column generation process we use $\alpha = 0.0$ and $m = |D|$ for a small number of iterations; in a second phase, we use $\alpha = 0.8$ and $m = |D|$, again for a small number of iterations; finally, we use $\alpha = 0.3$ and $m = |D|/4$ until the algorithm terminates.

With these settings, we start by generating a set of disjoint schedules at every iteration, since if the dual value of a ride is discounted to 0 then there is no incentive to include that ride in any further generated schedule. This gives a balanced coverage of the rides. Since the drivers are randomly reordered before each column generation iteration, the drivers have a diverse set of schedules: the rides with the greatest prizes are assigned to different drivers. Following iterations generate a set of highly overlapping schedules. These schedules tend to cover rides with large negative dual values

several times and in some sense provide a good coverage of the difficult rides. In the remainder of the column generation process, we generate schedules without much overlap, for only a small set of drivers. This last phase is intended to fill the gaps.

In our experience, these techniques decrease the number of major iterations needed to solve LP to near optimality by 30-50%. This also means that the size of each resulting formulation LP' is noticeably smaller and therefore is faster to solve.

When computing lower bounds as discussed in Section 3.2, we do not discount ride duals. In the fixing phase discussed in Section 4 we use $\alpha = 0.8$ and $m = |D|/8$.

3.3.3 Treating car constraints as cutting planes

Most of the non-zeros in the formulation appear in the car constraints. Recall that a column has the following form:

$$\text{column}^T = [\underbrace{00001000}_{1 \text{ driver}}, \underbrace{0000100001000100000000}_{\text{few rides}}, \underbrace{00000111111111100000000000000000}_{\text{many periods}}]$$

To keep the size of the linear programming relaxation LP' under control we have treated the car constraints as “cutting planes.” This is a well-known integer programming technique. Initially all car constraints are relaxed. If constraints for a car type are violated in all time periods in $[a, b]$ then we add to the formulation the constraint for time period $\lceil (a + b)/2 \rceil$. In practice we need to add only a handful of cuts; this significantly reduces the size of the formulation, and thus reduces the time needed to solve LP to near-optimality by about 20%.

3.3.4 Speeding up the shortest path computation

We next describe two simple ideas that helped reduce the time spent in shortest path computations approximately 20-fold. Although shortest path computation is very fast (linear time), these subproblems constitute a computational bottleneck due to their size. We may have a few hundred drivers, each with several possible shift start times, and a few car types; the product may yield as many as 2500 subproblems to solve in each iteration, each of which may have a graph with many hundreds of nodes (rides) and perhaps 500000 arcs (possible connections).

We have taken two steps to overcome this problem. First, we precompute as much as possible. Instead of building a graph separately for each subproblem, we build a master graph of the rides for each car type, including all mileage costs and car/ride mismatch penalties. For a given driver, shift start time, and car type, we only need to adjust the costs of the nodes and the costs of the arcs returning to the base t .

Second, we do exact column generation only every twenty major iterations. In the other iterations we work with a restricted connection graph in which there is no slack — time the driver spends waiting for the customer after arriving at the pickup location — greater than one hour. The general idea of not doing exact column generation in every iteration is a well-known integer programming technique.

4 Solving the integer program

In this section we describe a fix-and-price heuristic that finds feasible solutions to the integer program (1)-(6). See [4] for an in-depth discussion on solving large integer programs with column generation. The heuristic starts with the set of columns produced by Algorithm 1 and combines variable fixing with column generation to obtain a good solution. It is given a target optimality gap for the final solution value, and switches from fixing to column generation if the value of the current (restricted) LP becomes too large with respect to the optimum. We use the lower bound described in Section 3.2 to give a conservative estimate of the optimality gap. We use a target gap of 1%, which means that the aim is to find an integral feasible solution with value at most 1.01 times the lower bound computed in the initial column generation phase.

Our heuristic iteratively restricts the current formulation by fixing some variables to 1 and augments it with new columns that are consistent with earlier fixing decisions. It also has a limited “follow-on” fixing step which we describe later. The heuristic is outlined in Algorithm 2. This algorithm can also be viewed as a diving heuristic in which we explore only a single branch at each branch-and-bound node.

As discussed in Section 3, we mostly use the volume algorithm for solving the linear programs in the initial (column generation) phase. Though the volume algorithm produces good dual solutions for column generation, it produces primal solutions with few values near 1, which is undesirable for a fixing heuristic. In the fixing phase, we only use the simplex algorithm as it produces (extreme point) solutions which are “less fractional.”

Based on the solution of the initial LP, produced in the column generation phase, we fix all columns with values 0.99 or more to 1. In addition, ride-bump variables with value 0.95 or higher are set to 1. This is the only step in the algorithm in which we fix the values of the ride-bump variables explicitly. After this step the value of the LP typically is unchanged from its value at the start and 5-10% of the rides are covered with the fixed columns. The number of rides bumped after this step is usually less than 1% of the total, and on average half of the rides bumped in the final solution are identified in this step. After this initial fixing step, we reduce the size of the formulation (see Section 4.1), execute follow-on fixing (Section 4.2), and start fixing columns iteratively (Section 4.3).

4.1 Reducing the size of the formulation

Whenever a column is fixed to 1, other columns that have the same driver or some of the same rides become redundant due to constraints (2) and (3). Furthermore, constraints (2) and (3) themselves become redundant since no other column in a feasible solution can have the same driver or contain the rides served by that driver. Therefore, throughout the heuristic, whenever a column is fixed to 1 we reduce the size of the formulation by deleting the redundant rows and columns. We also suppress generation of columns in the future for a driver whose schedule has been fixed and we modify the column generation algorithm to avoid generating schedules for other drivers that serve the rides already assigned to the driver whose schedule has been fixed. Similarly, we delete all columns that include bumped rides and modify the column generation algorithm to avoid generating new schedules that contain them.

As discussed earlier in Section 3, we use a relaxation of (2)-(4) obtained by changing the ride

Algorithm 2 Fix-and-Price Heuristic

- 1: Re-solve the LP using Simplex
- 2: Fix some variables to 1
- 3: Reduce the size of the formulation
- 4: **for** some number of iterations **do**
- 5: Fix follow-on rides
- 6: **if** LP solution value has deteriorated significantly **then**
- 7: Generate more columns and re-solve the LP
- 8: **end if**
- 9: **end for**
- 10: Generate more columns and re-solve the LP
- 11: Delete (non-basic) columns with large reduced cost from the formulation
- 12: **repeat**
- 13: Fix some of the variables to 1
- 14: Reduce the size of the formulation and re-solve LP
- 15: **if** the LP value has deteriorated or most of the drivers are already fixed **then**
- 16: Generate more columns and re-solve the LP
- 17: **end if**
- 18: Delete columns with large reduced cost from the formulation
- 19: **until** the solution is integral
- 20: Return solution

constraints (3) to inequality. The reformulation step also guarantees that rides are covered at most once in the final solution.

4.2 Follow-on fixing

After reducing the size of the formulation, we identify pairs of rides that appear consecutively with large weight in the (fractional) optimal solution of the LP. More precisely, we compute a weight $fon(r_1, r_2)$ for every ordered pair of rides (r_1, r_2) that can be scheduled one after the other in a feasible schedule. Let $PAIR_{ij}$ be the set of all pairs of consecutive rides in schedule s_{ij} . Define:

$$fon(r_1, r_2) = \sum_{i \in D} \sum_{\substack{j \in S_i: \\ (r_1, r_2) \in PAIR_{ij}}} x_{ij}$$

If $fon(r_1, r_2)$ is greater than 0.95 for a pair of rides we implicitly fix it to 1 by forcing schedules to either contain both of the rides consecutively, or neither of them. This is achieved by deleting non-conforming columns from the formulation and modifying the column generation algorithm to produce only columns that conform to this restriction. We then re-solve the restricted problem, and generate more columns if the value of the LP increases by more than 0.5%. We recompute the weights and repeat this procedure until there are no pairs of rides with fon weight greater than 0.95. Next, we continue follow-on fixing by choosing a small number of pairs with the largest fon weights. We terminate the procedure when the largest fon weight is below 0.85.

The fixing procedure typically reduces the total running time of the heuristic by 20% and does not affect the quality of the final solution. This idea has been used in airline crew scheduling problems and it was brought to our attention by J. Forrest [9]. Also note that follow-on fixing can be viewed as diving in an enumeration tree in which branching decisions are made using the Ryan-Foster branching rule [17].

4.3 Iterative fix-and-price procedure

After follow-on fixing, we generate more columns, attempting to reduce the gap between the current LP value and the lower bound to half the target optimality gap. Since this may not always be possible, we also limit the number of column generation iterations. We then delete all non-basic columns with a reduced cost larger than 10% of the actual — not reduced — cost of an average column.

Finally we start the iterative fix-and-price procedure which chooses a set of columns \bar{S} that satisfies $\sum_{i,j \in \bar{S}} (1 - x_{ij}) < 1$ and fixes all of the columns in \bar{S} to 1. This condition also guarantees that the resulting restricted formulation does not become infeasible (it is possible to obtain a feasible integral solution by deleting all columns that have not been fixed). We include in \bar{S} the columns with the largest values. The idea is to fix many columns to 1 if they are all close to 1 in the current solution, and fix just a few (possibly only one) if all values are far from 1.

After fixing the columns in \bar{S} to 1, we reduce the size of the formulation as discussed in Section 4.1 and re-solve the resulting LP. If the gap between the LP value and the lower bound exceeds the target optimality gap, we generate more columns. If most (we use 70%) of the driver schedules have already been fixed to 1, we generate more schedules regardless of the LP value. In earlier experiments, we realized that both the column generation and the LP reoptimization take very little time once most of the schedules are fixed, since the LP size has been substantially reduced. In practice, generating more columns at this stage can decrease the optimality gap from the target gap (1%) to a significantly smaller value (0.5%). Finally, we delete from the formulation all non-basic columns with large reduced cost (at least 50% of the average cost of a column) and repeat the iterative fixing procedure.

When the iterative fixing procedure terminates, the solution of the current LP formulation is integral.

4.4 Computational performance

We measured the performance of the fix-and-price heuristic on six real-world sample data sets. Table 1 shows the size of the instance, its computation time (minutes:seconds), the optimality gap (based on the lower bound) and number of columns generated to solve the linear and integer problems to near-optimality. The computations are done on a single processor of a current-generation IBM RS6000 workstation.

All of these problems were solved to near-optimality in under 3 minutes (much less than the 15 minutes time limit, as turned out to be the case in practice). Most of the time is spent on obtaining a high-quality LP solution. We can then generate an integral solution very quickly. During the

	Number of		LP			LP+IP		
Instance	Rides	Drivers	Time	Gap	Cols	Time	Gap	Cols
Prob 1	685	171	1:40	0.10%	3494	2:58	1.00%	5165
Prob 2	672	158	1:37	0.09%	3425	2:25	1.22%	5065
Prob 3	565	136	1:16	0.07%	2950	1:31	0.58%	3521
Prob 4	516	187	1:16	0.07%	2943	1:25	0.89%	3218
Prob 5	684	208	1:40	0.06%	3907	2:18	0.76%	5499
Prob 6	492	191	1:07	0.07%	2891	1:16	0.77%	3147

Table 1: Sample Runs

solution process, we generate 15 to 30 schedules per driver.

4.5 Solving the integer program in continual mode

There are two issues that influence how the integer program is handled in continual mode. The first is that lateness is allowed, i.e., a driver is allowed to arrive late to pick up a passenger, though a penalty must be paid for being late. See [20] for a similar application from the airline industry. The second issue is that in continual mode, we are working under a time constraint; we have to deliver a solution reasonably quickly.

Lateness affects solving *LP* (the full formulation) and thus the integer program indirectly: the connection graph is more dense and many more columns can be generated. Therefore solving any individual column generation subproblem is significantly slower than without lateness. In the next two subsections, first we describe how we address the lateness problem in generating columns, then how we modify the fix-and-price algorithm to stay within the allotted running time (as well as address other arising issues).

4.5.1 Column generation with lateness

In continual mode there is a maximum allowable start time for each ride. Up to this limit the ride may be served late, but a penalty is incurred; this penalty increases with increasing lateness. In this case, the pricing problem can still be formulated as a simple shortest path problem as follows, but in a graph larger than the one described in Section 3.1. Also see [20].

Now there is a directed arc (r_1, r_2) from ride r_1 to ride r_2 if, after serving ride r_1 , it is possible to reach the start location of ride r_2 no later than the maximum allowed lateness for r_2 . Since the granularity of the data is 1 minute and the maximum allowed lateness is bounded by a constant we could, in principle, create multiple nodes for each ride (one for each possible start time) and still have a linear time algorithm (in the number of arcs, which is quadratic in the number of rides) for

generating columns. However, the constant factor makes this prohibitively expensive. Instead, we have adopted the following strategy.

Two restricted column generation procedures are used. The first allows a late arrival for a ride, but the ride must be finished on time. This is possible because for many rides an “early arrival buffer” is specified. The target time for the driver to arrive at the pickup location is earlier than the passenger’s reservation time, but the ride is assumed to start at the reservation time. We may use this margin to schedule a late arrival. Note that in this approach we further limit the lateness of the driver’s arrival to the length of the “early arrival buffer”, which may be less than the allowed lateness. By doing this, we may disallow some legal schedules in order to have a smaller graph.

The second method is significantly slower, but it considers a larger set of schedules. However it, too, may exclude some legal schedules. We create only one node for each ride and include arc (r_1, r_2) if r_2 can be served after r_1 (albeit possibly late) and the original start time of r_1 is no later than that of r_2 ; this keeps the graph acyclic. Otherwise, with long lateness and short ride time there could be a cycle, though it is unlikely. As we proceed in the shortest path computation we attach two labels to each node, its cost and its lateness. When a node is labeled with a cost, it is labeled with the cheapest possible way (including penalties) to reach it within its lateness limit among paths through earlier nodes (whose latenesses are already fixed). This determines the lateness for the current node as well as the cost.

With this restricted column generation we are solving a restricted linear programming problem thus we generate a lower bound only for the restricted problem, not the original *LP*. Nonetheless, in practice this approach works remarkably well: experiments demonstrate that this lower bound is not much higher than a true lower bound computed using column generation for the original problem.

4.5.2 Changes in fix-and-price in continual mode

In continual mode, there is a limit (say, 15 minutes) on the total execution time of the integer programming module. To guarantee that our algorithm terminates in the allotted time (preferably with a feasible solution) we do the following:

1. In most column generation iterations, use the first method (i.e., with lateness limited to the early arrival buffer time);
2. terminate generating columns in the initial (LP) phase if 40% of the allocated time has passed;
3. generate fewer columns in the second (IP) phase if 65% of the allocated time has passed;
4. delete most of the non-basic columns from the formulation and start generating very few new ones after 85% of the allocated time has passed;
5. terminate the algorithm without a feasible solution if the time is up.

Though the basic building block of our approach, column generation, is slower in continual mode because of lateness allowances, there are a few aspects of continual mode that speed up our computations. A driver that has started his schedule is already assigned a car type and his start time is fixed. As the day progresses, more and more rides (those that are in the past) will be locked to his schedule. Restricting the column generation algorithm to produce schedules consistent with these

constraints simplifies the pricing problem by reducing the size of the solution space. In addition, an initial solution is available in continual mode: the schedule produced by the previous call to the optimizer and updated by the heuristic solver to accommodate changes in the data. In this case, we simply add the columns corresponding to these individual driver schedules to the formulation at the beginning of the column generation phase. This typically speeds up the convergence of the column generation algorithm.

4.6 Online solution quality

In this section we compare the offline schedule cost at the beginning of the day with the online cost seen at the end of the day for the six sample data sets introduced in Section 4.4. It would be more natural to compare the online cost with the optimal cost on the final, end-of-day data, but unfortunately this is not possible. The larger system our scheduler resides in encodes various events as artificial ride requests and artificial modifications to the true ride requests data. For example, a driver positioned at an airport in the expectation that demand may emerge there, is dispatched on a series of artificial, zero-distance rides from and to the airport. Similarly, if a ride is supposed to start at 12:00 but the passenger arrives at 12:30, the ride is re-encoded as one with a start time of 12:30. These artificial re-encodings capture the data essential to the optimizer for computing a valid schedule for the remainder of the scheduling day, but the resulting end-of-day data has many artificial values mixed in with the true demand data. We do not have access to the system to extract the actual data, nor any way to distinguish real and artificial data values in the log files.

A comparison of online to optimal performance on the end-of-day data, even if it were possible, would still have limitations. We are not simply comparing our online algorithm to a clairvoyant one, but also comparing observed travel times to estimated ones. Even the end-of-day data cannot be clairvoyant in this sense: actual travel times for segments driven are measured, but travel times for all other possible segments are available only as projections. Also, the online performance is not just that of our scheduler, but that of a system including human dispatchers who make the final scheduling decisions.

In Table 2, we compare actual online end-of-day cost to offline start-of-day projected cost. Because the sets of demands are not the same, it does not make sense to compare total schedule costs. Instead we present the data in terms of the average cost per ride (CPR), defined as the total driver wages and mileage costs divided by the number of rides. For the six sample instances, the online CPR is 7–28% greater than the start-of-day offline CPR.

5 Local-Search Heuristics

In this section we describe the local-improvement heuristic for the SSS. In *instantaneous* mode, if for example a ride runs long, making the driver’s next ride infeasible, bumping the infeasible ride gives a schedule that is feasible but suboptimal. There is no time to invoke IP to reoptimize, and the local solver is used to try to reschedule the ride in a near-optimal fashion, in an allotted time of 15 seconds. In this case the local heuristics are invoked with special attention to the

Instance	Rides before	Cancelled	New	Rides after	CPR Increase
Prob 1	685	29	103	759	16%
Prob 2	672	41	95	726	13%
Prob 3	565	31	52	586	28%
Prob 4	516	23	113	606	7%
Prob 5	684	54	61	691	22%
Prob 6	492	23	50	519	19%

Table 2: Start-of-day offline vs. end-of-day online cost per ride (CPR) comparison

bumped ride (which should be assigned to a driver if possible) and the driver it came from (whose already-changed schedule is ripe for the addition or substitution of other rides). Also, when a *continual-mode* IP reoptimization completes, the new schedule must be made consistent with the by-now changed data. In this case several rides and drivers — all those whose schedules cannot be as in the IP solution — are singled out for special attention.

The local solver may also be called upon in rare circumstances to produce a schedule from scratch to solve the offline problem. Experiments on the examples of Table 1 show that with a 2-minute run (the time consumed on average by IP), the local solver’s offline schedules are on average under 6% worse. This declines with increasing run time to under 3%; details are given in Section 5.5.

5.1 Problem formulation and complications

The local-search solver starts with the same inputs as described for the IP solver, along with an initial solution. (In offline mode, the initial solution simply has all rides on the bumped list, and no cars assigned to any driver.) Optionally but typically, a list of drivers, rides, or both is provided as hints; these are to be given special consideration for schedule improvement.

The same cost function as that used by the IP solver is used to drive local improvement. The cost of a small change (such as rescheduling a single ride) can be evaluated quickly by updating the costs for just its original and new drivers.

Unlike the IP solver, the local solver must deal with infeasible states. We developed a set of policies for managing infeasibility. An infeasible driver schedule is given a fixed, prohibitively high cost. Since adding further rides to such a schedule would decrease other costs without increasing this fixed cost, seemingly improving matters but actually making them worse, the local solver ignores infeasible driver schedules whenever possible. (However, if called with special attention to a driver whose schedule is infeasible, the local solver does attempt to minimize its cost, and in particular to make it feasible.) The local solver guarantees not to make any feasible driver schedule infeasible.

5.2 Depth-one and depth-two ride-assignment search

We now introduce the local solver’s methods, and later revisit how they are employed, but for the moment imagine that the goal is simply to schedule a single bumped ride r . Recall that R and D are the sets of all rides and drivers; we will use R^* and D^* to be subsets of special interest.

The most-used primitive action is to move a ride from its current driver (or the bumped list) to a specified driver (or the bumped list) and revise the schedule cost. Using this primitive, depth-one search $\text{Insert}(r, D^*)$ tries all such possibilities and selects a cheapest one. Simply invoking $\text{Insert}(r, D)$ may be good enough for inserting a single new ride, or (applied sequentially to all rides, perhaps in order of their pickup times) for constructing a plausible initial schedule from scratch, but it can easily get trapped in a local minimum: it is unable even to swap two roughly concurrent rides between a pair of driver schedules.

Depth-two search, $\text{InsertBump}(r, D_1^*, D_2^*)$ tries moving r to each schedule in D_1^* , but uses a different “move” primitive which bumps any conflicting rides; each such ride r' is then reassigned using $\text{Insert}(r', D_2^*)$. A simple swap can occur as follows. Suppose that a ride r is originally on driver d ’s schedule, and some ride r' is on another schedule d' . If InsertBump causes r to be moved to d' , displacing r' , and the subsequent Insert moves r' to d , then r and r' have traded places.

Depth-two search was found to be adequate for providing a reasonable schedule update in response to a small change such as a new ride. If a driver’s car type is not already locked, any or all of the ride-moving and insertion heuristics may also be allowed to choose a good car type.

The number of rides on a driver schedule (ten or so at most) and the number of car classes can be treated as small constants, i.e., $O(1)$, and so depth-one insertion takes time $O(n)$ (where n is the number of drivers) and depth-two insertion time $O(n^2)$, in typical use where the driver lists contain all drivers, $D_1^* = D_2^* = D$. The run time of depth-two search is large enough that it is an *interruptible* operation: if its allotted time runs out, it returns the best solution it has found so far.

5.3 From local towards global

Our final heuristic tool, *Ripup-redo*, has a very simple strategy: It unschedules a random set of, say, ten rides, and reschedules them using Insert . It selects the better of this solution and the one at the start of the round, and repeats this until a fixed number of rounds is reached or until it runs out of time, whichever comes first. The treatment of several rides at a time allows the local solver to break out of a local minimum, from which no single Insert or InsertBump move would give an improvement. Even if a local improvement is possible, we may not have time to find it; *Ripup-redo* offers a good tradeoff between runtime and solution quality.

In instantaneous-mode use, with just a few seconds to run, *Ripup-redo* often gives small improvements in the schedule, helping to keep it near optimal. In offline mode (in case of IP failure), given a couple of hours to run (much more than IP actually takes), *Ripup-redo* is the workhorse for constructing a schedule of near-IP quality.

In contrast to the procedure described above, we might accept the result of a round which *increases* the cost, in the hope of “climbing out of” a local minimum and finding a better solution in a later round, perhaps using simulated annealing [11] to control the increases. In fact we did not use such

an approach, because it would introduce control parameters that would have to be tuned for the full range of problem instances, and because the results without it seemed good enough.

While details such as using simulated annealing or not, and using `Insert` or `InsertBump`, are fairly arbitrary, others are fixed. For one, it is important that the ripped-up rides are reinserted in random sequence. If lower-numbered rides were always inserted first, for example, they would always be favored, and `Ripup-redo` would be less likely to hit on a globally preferable schedule.

Also, the number of rides rescheduled in each iteration, or “rip size”, must be chosen reasonably well. If it is too small then the iteration is likely to reconstruct the original schedule: the ripped-up rides are likely to be all at different times, and if the schedule is fairly full each will get rescheduled to its vacated original slot. If the rip size is too large then a round’s greedy scheduling of the ripped-up rides is overwhelmingly likely to produce a schedule worse than its starting schedule, and the round will return its original schedule. Thus it is important to pick a rip size not too large, but large enough that it is likely to include two rides at similar times. This suggests a rip size on the order of the square root of the “number of time slots.” Dividing our 24-hour schedule into 15-minute intervals (a reasonable level of resolution for this purpose) suggested about 100 time slots, and a rip size on the order of 10; experimentation confirmed this as a reasonable choice.

5.4 Use of heuristics to respond to an event

The foregoing operations are bundled up into a heuristic `Respond(R^* , D^*)`, which is called in response to triggering events. In instantaneous-mode use, if a ride r is bumped because its predecessor on driver d ’s schedule ran late, then `Respond($\{r\}$, $\{d\}$)` is invoked: only the ride and driver just changed are of special interest. In use for reconciling the foreground schedule and a schedule output by IP, trivially feasible changes are first made to the IP schedule to make it feasible with respect to the current input. New rides are placed on the bump list. Rides assigned in the IP schedule that conflict with committed actions in the foreground are likewise bumped. Then `Respond` is called on this amended IP schedule, with D^* being those drivers whose schedules differ from the IP output, and R^* those rides scheduled to different drivers (and all bumped rides). Pseudo-code of `Respond` is as follows:

Algorithm 3 Respond (R^* , D^*)

- 1: **for** each $r \in R^*$, `Insert(r , D)`
 - 2: **for** each $r \in R$, `Insert(r , D^*)`
 - 3: **for** each $r \in R^*$, `InsertBump(r , D , D)`
 - 4: `Ripup-redo(R , D , 10, 100)`
-

`Respond` is purely heuristic. The intent is to first try to schedule the special-interest rides before time runs out, then see if things can be improved quickly by focusing on the special-interest drivers (who may, for example, have gaps in their schedules). Next, if time remains we try to do a better job on the special-interest rides, and finally use any remaining time to try to improve the schedule generally.

In the unlikely event that the IP offline mode fails, the heuristic offline solver begins by calling `Respond` with all rides and no drivers (taking seconds), then runs `Ripup-redo` for the rest of the

allocated time (2 hours, for example).

5.5 Performance of the local-search solver

Performance of the heuristic solver is most easily quantified for the offline mode. Figure 1 shows the performance achieved by the local-search solver when solving from scratch on our six sample instances.

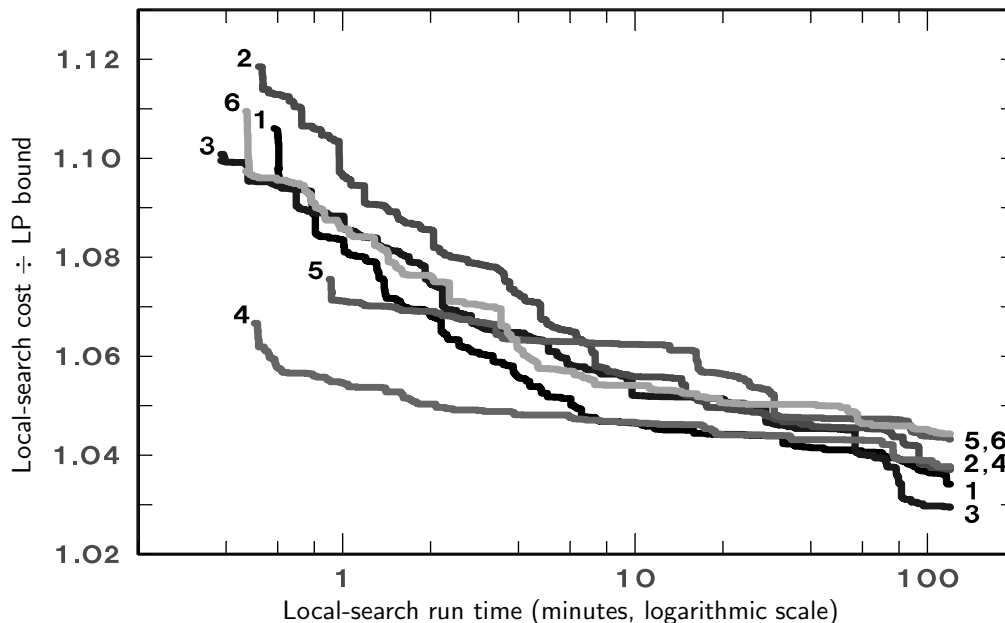


Figure 1: Cost of local-search solution as a function of run time, divided by cost of LP lower bound. Instance number from Table 1 is indicated to the left of each trace.

The local solver’s initial greedy solutions, produced in under a second, are almost 20% worse than the LP lower bounds; they are not shown in the figure. Application of the depth-2 search heuristic to all rides takes on average 33 seconds and reduces the average gap to under 10%; these solutions are the figure’s initial points. Ripup-redo progressively reduces the cost: after 20 minutes the average gap is under 5%, and after 90 minutes under 4%. Compared with the IP solutions, by the 2-minute mark (the average time taken for IP) the average difference is under 6%, in 5 minutes it is under 5%, in 25 minutes under 4%, and in 2 hours under 3%.

6 Conclusion

In this paper we have described an optimization-based approach to a highly dynamic real-time problem. We periodically solve static snapshots of the dynamic problem to near-optimality. We are able to solve the snapshot quickly enough that the solution can be used with minor modifications. In parallel, we dynamically maintain a schedule that is always feasible with respect to the up-to-

the-minute input and is much more efficient than manually constructed schedule. Our system is in full-time use and productivity has increased significantly.

Other service operations such as mobile installation and repair service and package delivery share the key characteristics of the problem that made this solution possible and profitable: a complex optimization problem that can be solved to near-optimality, and constantly changing demands. We expect this type of continual optimization to become more common in the future. Hopefully, this will lead to more progress in IP and heuristic solution techniques.

Acknowledgments

We thank our colleagues at IBM Research: Vernon Austel for his help in the implementation, and Francisco Barahona and John Forrest for very fruitful technical discussions.

References

- [1] F. Barahona and R. Anbil, On some difficult linear programs coming from set partitioning, *Discrete Applied Math.* **118** 3–11 (2002).
- [2] F. Barahona and D. Jensen Plant location with minimum inventory, *Mathematical Programming* **83** 101–111 (1998).
- [3] C. Barnhart, N.L. Boland, L.W Clarke, E.L. Johnson, G. L. Nemhauser, and R.G. Sheno, Flight String Models for Aircraft, Fleeting and Routing, *Transportation Science* **32** 208–220, (1998)
- [4] C. Barnhart, E. L. Johnson, G.L. Nemhauser, M.W.F. Savelsbergh and P.H. Vance, Branch-and- Price: Column Generation for Solving Huge Integer Programs *Operations Research* **46** 316–329, (1998).
- [5] J. Bramel, and D. Simchi-Levi. On the effectiveness of set covering formulations for the vehicle routing problem with time windows. *Operation Research*, 45(2):295–301, March 1997.
- [6] J. Desrosiers, Y. Dumas, M.M. Solomon, and F. Soumis. Time constrained routing and scheduling, In *Network Routing*, pp. 35–139, M. O. Ball, T. L. Magnanti, C. L. Monma, and G. L. Nemhauser (editors). NorthHolland, (1995).
- [7] J. Desrosiers, F. Soumis, and M. Desrochers, Routing with time windows by column generation, *Networks* **14** 545–565 (1984).
- [8] M. L. Fisher. Vehicle Routing. In *Handbooks in Operations Research and Management Science*, volume on Network Routing, M. Ball, T. Magnanti, C. Monma, and G. Nemhauser, editors, 1–33, 1995.
- [9] J. Forrest, personal communication.
- [10] P. Gilmore and R. Gomory, A linear programming approach to the cutting stock problem – Part II, *Operations Research* **11** 863–888 (1963).
- [11] S. Kirkpatrick, C. D. Gelatt, Jr., and M. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.
- [12] S. O. Krumke, J. Rambau, and L. M. Torres. Real-time dispatching of guided and unguided automobile service units with soft time windows. In *Proc. 10th Annual European Symposium on Algorithms*, Lecture Notes in Computer Science, Springer, 2002.
- [13] L. Lasdon, *Optimization theory for large systems*, McMillan, London (1970).

- [14] M. Lubbecke and J.Desrosiers, Selected topics in column generation, technical report, *Les Cahiers du GERAD G-2002-6* (2002).
- [15] J. Nazareth, *Computer solution of linear programs*, Oxford University Press, Oxford, (1987).
- [16] B. Rexing , C. Barnhart, T. Kniker, A. Jarrah and N. Krishnamurthy, Airline Fleet Assignment with Time Windows. *Transportation Science*, 34:1, 1–20, 2000.
- [17] D. Ryan and B. Foster, An Integer Programming Approach to Scheduling, In *Computer Scheduling of Public Transport Urban Passanger Vehicle and Crew Scheduling*, pp. 269-280, A. Wren (editor). North-Holland (1981).
- [18] M. M. Solomon. On the worst case performance of some heuristics for the vehicle routing and scheduling problem with time windows constraints. *Networks*, 16:161–174, 1986.
- [19] M. M. Solomon, and J. Desrosiers. Time window constrained routing and scheduling problems. *Transportation Science*, 22:1–13, 1988.
- [20] M. Stojkovic, F. Soumis, An Optimization Model for the Simultaneous Operational Flight and Pilot Scheduling Problem, *Management Science* **47** 1290–1305 (2001).