REAL-TIME DISPATCHING OF GUIDED AND UNGUIDED AUTOMOBILE SERVICE UNITS WITH SOFT TIME WINDOWS

SVEN O. KRUMKE*, JÖRG RAMBAU*, AND LUIS M. TORRES**

ABSTRACT. Given a set of service requests (events), a set of guided servers (units), and a set of unguided service contractors (conts), the vehicle dispatching problem VDP is the task to find an assignment of events to units and conts as well as tours for all units starting at their current positions and ending at their home positions (dispatch) such that the total cost of the dispatch is minimized.

The cost of a dispatch is the sum of unit costs, cont costs, and event costs. Unit costs consist of driving costs, service costs and overtime costs; cont costs consist of a fixed cost per service; event costs consist of late costs linear in the late time, which occur whenever the service of the event starts later than its deadline.

The program ZIBDIP based on dynamic column generation and set partitioning yields solutions on heavy-load real-world instances (215 events, 95 units) in less than a minute that are no worse than 1% from optimum on state-of-the-art personal computers.

1. INTRODUCTION

The german automobile club *ADAC* (*Allgemeiner Deutscher Automobil-Club*), the second largest automobile club worldwide, maintains a heterogeneous fleet of over 1600 service vehicles in order to help people whose cars break down on their way. All service vehicles (*units*, for short) are equipped with GPS, which helps to exactly locate each unit in the fleet. In five ADAC help centers (*Pannenhilfezentralen*) spread over Germany, human operators (*dispatcher*) constantly assign units to incoming help requests (*events*, for short) so as to provide for a good quality of service (i.e., waiting times of less than 20–60 minutes depending on the system load) and low operational costs (i.e., short total tour length and little overtime costs). Moreover, about 5000 units of service contractors (*conts*, for short)—not guided by ADAC—can be employed to cover events that otherwise could not be served in time. This manual dispatching system is now subject to automatization.

Given a snapshot in the continuously running planning process, the task of the dispatcher in one of the help centers is to assign a unit or a contractor to each event and a tour to each unit such that every event is served by a unit or contractor that is capable of this service and such that a certain cost function is minimized. The result of this planning process is a (tentative) dispatch. The overall goal is to design an automatic online dispatching system that guarantees small waiting times for events and low operational costs when regarded over a larger period of time. In particular, the ADAC chose to impose a soft deadline on the service of an event that may be missed at the cost of a linearly increasing lateness penalty (*soft time windows*). Figure 1 indicates a

*Supported by the DFG research center "Mathematics for key technologies" (FZT 86) in Berlin.".

**Supported by the German Academic Exchange Service (DAAD, grant A/99/03594).

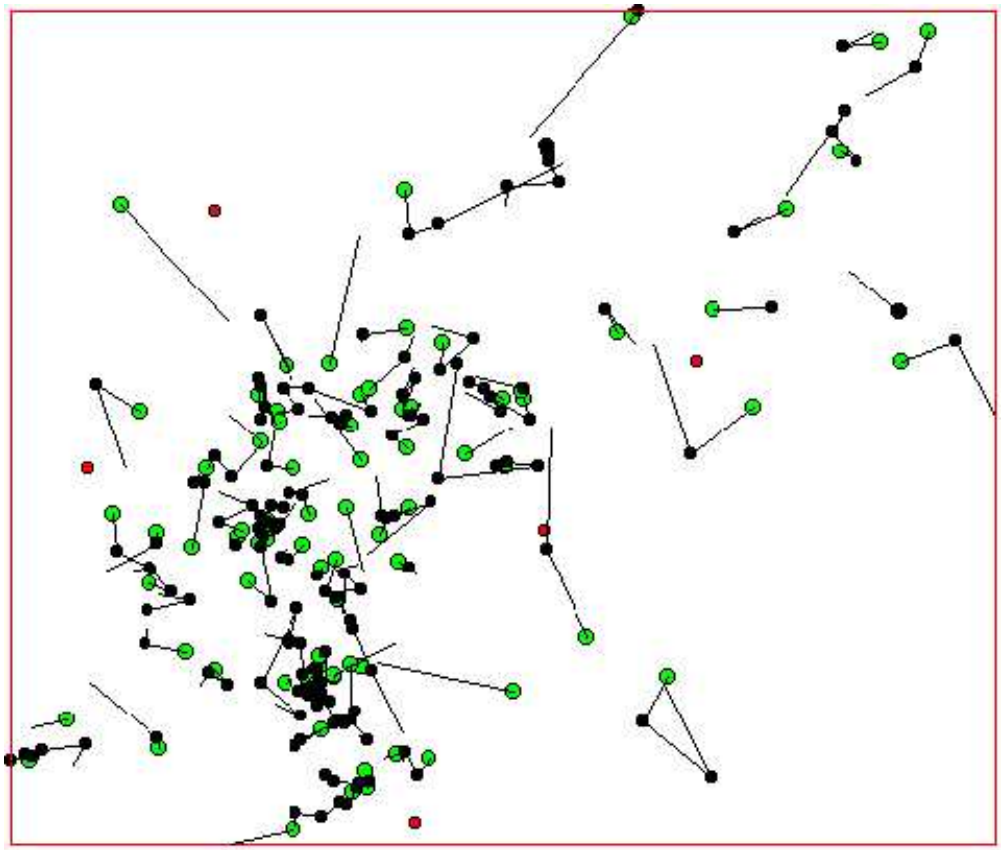


FIGURE 1. A real-world instance of VDP (gray: units, black: events). Lines indicate tours that end at the home positions of units in some dispatching solution

typical, obviously not uniform distribution of units (gray) and events (black) in a real world instance of ADAC.

A basic building block for such a system is a fast *offline optimization module* that is able to produce an optimal or near-optimal dispatch under strict real-time requirements. More specifically, the optimization module must provide a reasonable answer in less than a second and must have the ability to improve on that solution whenever by some circumstances more time is granted to the optimization process. Given the fact that in an average snapshot 200 yet unserved events have to be assigned to tours for about 100 vehicles at distinct positions, the real-time aspect requires special attention. See [1] for evidence that the ability of fast reoptimization can help to improve the performance of dynamic dispatching systems. A possible method to organize the dynamics of a dispatching system in the language of *agents* can be found in [14]. Whether or not precomputed routes should be subject to sudden change are discussed in [5], where the optimization part is done by tabu search.

The basic problem can be modeled as a *multi depot vehicle routing problem with soft time windows* MVRPSTW, where each guided vehicle is a depot of its own and

the contractors maintain a depot of a certain capacity. Many algorithms, heuristic and exact (i.e., where a performance guarantee can be given a-posteriori), have been proposed for the related *vehicle routing problem with time windows*, where the time windows have to be respected in any feasible dispatch (see [3] and references therein for a survey of various problem types and exact algorithms; see [10] for recent progress in efficiency of exact algorithms; see [13] for a tabu search approach dealing with soft time windows; see [11] for one approaches based on genetic algorithms).

To the best of our knowledge none of the exact algorithms was ever reported to *predictably* meet strict real time requirements in a large-scale real-world application, i.e., produce reasonable answers very early (after five seconds) in the course of the optimization process. On the other hand, the heuristic methods cannot guarantee a certain quality of the delivered solution. What should be done in practice? (The primary opinion in industry when being asked to present an approach to solve this problem was to use meta-heuristics because exact approaches would not be able to produce solutions fast enough.)

We will show in this paper that we can employ a custom-made dynamic column generation method to this problem that delivers good solutions after a fraction of a second and yields a provably optimal or near-optimal solution in less than five seconds in all real-world data sets with about 200 events and about 100 units provided by ADAC. The behaviour remains stable even for extremal load problems (artificially augmented real-world problems) with up to 770 events and 200 units: the solution quality was already within 12% from optimum after 5 seconds, within 5% from optimum after 15 seconds, and within 2% after one minute.

We had the chance to compare an implementation of our algorithm with an experimental prototype using meta-heuristics based on genetic algorithms and hill-climbing, which was produced by our industrial partner with serious effort; we show that four variants of the code based on meta-heuristics are clearly outperformed by our exact method.

Using the exact approach is viable mainly because we can explicitely exploit a rather obvious but nevertheless crucial structure in the real-world problem data provided by ADAC: Since in a planning snapshot all events have been released already, the relatively tight soft time windows will *enforce optimal solutions with almost only short tours*.

The problem for which we propose a solution in this work is how to specifically make use of this advantageous property to accelerate convergence of a column generation procedure for our dispatching problem. Our solution to this is *Dynamic Pricing Control*.

The issues that need to be addressed in order to gain speed in convergence in a dynamic column generation method are

- reduce the number of columns generated
- consolidate the dual variables as early as possible
- accelerate the solution of the pricing problem

Other issues that on other problem sets may cause problems—like finding an integer solution—have turned out to be well-behaved for our data sets: we could, e.g., get away by using the MIP solver of CPLEX 7.0 to find good integer solutions in the course of the column generation. This also means that there is room for improvement in our

algorithm by employing a genuine branch&price algorithm. From the practicioner's point of view, however, the integrality problem in the ADAC data sets is solved by CPLEX "off the shelf".[1]

In order to emphasize that the structure of our dispatching problem is special, we will use the name *vehicle dispatching problem* (VDP, for short) for the problem studied in this paper.

The rest of the paper is organized as follows: In the next section we introduce the exact setting of the VDP. Section 3 is devoted to the mathematical model that is the basis for the column generation approach. In Section 4 we describe our real-time compliant algorithm. Computational results on real-world data in Section 5 prove the performance of the algorithm, where Section 6 evaluate the effectiveness of various algorithmic tuning concepts. Section 7 summarizes the key points of this paper.

2. PROBLEM SPECIFICATION

In the following, we specify the exact form of VDP that is tackled by our algorithm. An instance of the VDP consists of a set of units, a set of contractors, and a set of events.

Each unit u has a current position o_u, a home position d_u, a logon time t_u^{start}, a shift end time t_u^{end}, and a set of capabilities F_u. Moreover, the costs related to using this unit are specified by values for costs per time unit for each of the following actions: driving c_u^{drv}, waiting c_u^{wait}, serving c_u^{svc}, and overtime c_u^{ot}.

Each contractor v has a home position d_v and a set of capabilities F_v. Moreover, the costs for booking the contractor are specified by a value for costs per service c_v^{svc}.

Each event e has a position x_e, a release time θ_e^{r}, a deadline θ_e^{d}, a service time δ_e, and a set of required capabilities F_e. Moreover, extra costs related to serving this event are specified by the value of a lateness coefficient c_e^{late} meaning that a cost of c_e^{late} times the delay w.r.t. the deadline of the event is incurred.

A feasible solution of the VDP (a *dispatch*) is an assignment of events to units and contractors capable of serving them, as well as a tour for each unit such that all events are assigned, the service of events does not start before their release times (waiting is allowed), and all tours for all units start at their current positions not before their logon times and end at their home positions. The costs of a dispatch are the sum of all unit costs, contractor costs, and event costs.

3. MODELING

We will use a model based on tour variables. Models of this type are by now well-established in the vehicle routing literature (see, e.g., [3]).

Let $\mathcal{R}$ be the set of all feasible *tours*. This set splits into the sets $\mathcal{R}_u$ of feasible tours for each unit u. A tour in $\mathcal{R}_u$ can be described by an ordered sequence $(u, e_1, e_2, \ldots, e_k)$ of k distinct events visited by u in that order. We will use the sequence (u) to denote the *go-home* tour, i.e., the tour in which u travels from its current position directly to its home position. Feasibility means that the capabilities of the unit are sufficient for e_i, i.e., $F_{e_i} \subseteq F_u$, for all $i = 1, \ldots, k$. Notice that this sequence also fixes the arrival times of u at each event.

[1] One has to take into account that the chance for the incorporation of an algorithmic method in an industrial product heavily depends on the relation between effort and benefit.

For all $R \in \mathcal{R}_u$ we introduce binary variables x_R with the following meaning: $x_R = 1$ if and only if the route R is chosen to be in the dispatch.

The cost of the route R is denoted by c_R and computed as follows. Let δ_u^{ef} be the driving time of unit u from event e to event f. Moreover, let $\delta_u^{o_u e}$ resp. $\delta_u^{e d_u}$ be the driving times of unit u from its current position to event e resp. from event e to its home position d_u. By t_R^e we denote the arrival time at event e in route R. The arrival time of u at its home position be $t_R^{d_u}$. Then the cost c_R of route $R = (u, e_1, e_2, \ldots, e_k)$ can be computed as

$$
\begin{aligned}
c_R = & \\
& c_u^{drv}\delta_u^{o_u e_1} + \sum_{i=2}^{k} c_u^{drv}\delta_u^{e_{i-1} e_i} + c_u^{drv}\delta_u^{e_k d_u} && \text{(driving)} \\
& + \sum_{i=1}^{k} c_u^{svc}\delta_{e_i} && \text{(service)} \\
& + c_u^{ot}\max\{(t_R^{d_u} - t_u^{end}), 0\} && \text{(overtime)} \\
& + \sum_{i=1}^{k} c_{e_i}^{late}\max\{(t_R^{e_i} - \theta_{e_i}^{d}), 0\} && \text{(lateness)}
\end{aligned}
$$

A feasible "route" S for a contractor v can be written as a set $\{e_1, e_2, \ldots, e_k\}$ of events that this contractor may be assigned to serve, i.e., $F_{e_i} \subseteq F_v$ for all $i = 1, \ldots, k$.

Let t_v^e be the time by which contractor v can have reached event e with one of his vehicles. The cost c_S of such a "tour" S can be computed as follows:

$$
\begin{aligned}
c_S = & \\
& c_v^{svc}|S| && \text{(service)} \\
& + \sum_{i=1}^{k} c_{e_i}^{late}\max\{(t_v^{e_i} - \theta_{e_i}^{d}), 0\} && \text{(lateness)}
\end{aligned}
$$

Since this cost is linear in the events served by this contractor, every "tour" S of a contractor can be combined from elementary contractor tours containing each only a single event.[2] Let $\mathcal{S}^v$ be the set of elementary feasible "tours" for v, and let $\mathcal{S}$ be their union over all contractors $v \in V$.

For all $S \in \mathcal{S}^v$ we introduce binary variables x_S with the following meaning: $x_S = 1$ if and only if the elementary "tour" S is chosen to be in the dispatch.

The VDP can now be formulated as a set partitioning problem as follows. Let a_{Re}, b_{Se} be binary coeffi cients with $a_{Re} = 1$ (resp. $b_{Se} = 1$) if and only if event e is served in tour R (resp. in elementary contractor "tour" S).

[2]So far, there are no data about the capacities of contractors available to the ADAC. Thus, an infinite capacity is assumed. Later, in the dynamic optimization process, a contractor that declines a request will be removed form the dispatching system for some time.

$$\min \sum_{R \in \mathcal{R}} c_R x_R + \sum_{S \in \mathcal{S}} c_S x_S \tag{IP}$$

subject to

$$\sum_{S \in \mathcal{S}} b_{Se} x_S + \sum_{R \in \mathcal{R}} a_{Re} x_R = 1 \qquad \forall e \in E; \tag{1}$$

$$\sum_{R \in \mathcal{R}_u} x_R = 1 \qquad \forall u \in U; \tag{2}$$

$$x_R \in \{0,1\} \qquad \forall R \in \mathcal{R}; \tag{3}$$

$$x_S \in \{0,1\} \qquad \forall S \in \mathcal{S}. \tag{4}$$

Our method is based on solving the linear programming relaxation (LP) of (IP), where the integrality constraints (3) and (4) are replaced by the non-negativity constraints (3') and (4'):

$$\min \sum_{R \in \mathcal{R}} c_R x_R + \sum_{S \in \mathcal{S}} c_S x_S \tag{LP}$$

subject to

$$\sum_{S \in \mathcal{S}} b_{Se} x_S + \sum_{R \in \mathcal{R}} a_{Re} x_R = 1 \qquad \forall e \in E; \tag{1}$$

$$\sum_{R \in \mathcal{R}_u} x_R = 1 \qquad \forall u \in U; \tag{2}$$

$$x_R \geq 0 \qquad \forall R \in \mathcal{R}; \tag{3'}$$

$$x_S \geq 0 \qquad \forall S \in \mathcal{S}. \tag{4'}$$

It is evident that not all columns of the coeffi cient matrix can be statically enumerated for our problem size. It is, however, by now established that dynamic column generation can be used [3].

After each iteration of the column generation procedure we have the optimal solution of the *restricted* LP, in which only tours from a subset $\widetilde{\mathcal{R}} \subset \mathcal{R}$ have been considered. (We assume that all elementary contractor tours in $\mathcal{S}$ have already been added.)

$$\min \sum_{R \in \widetilde{\mathcal{R}}} c_R x_R + \sum_{S \in \mathcal{S}} c_S x_S \tag{RLP}$$

subject to

$$\sum_{S \in \mathcal{S}} b_{Se} x_S + \sum_{R \in \widetilde{\mathcal{R}}} a_{Re} x_R = 1 \qquad \forall e \in E; \tag{1'}$$

$$\sum_{R \in \widetilde{\mathcal{R}_u}} x_R = 1 \qquad \forall u \in U; \tag{2'}$$

$$x_R \geq 0 \qquad \forall R \in \widetilde{\mathcal{R}}; \tag{3''}$$

$$x_S \geq 0 \qquad \forall S \in \mathcal{S}. \tag{4'}$$

Consider the dual of (RLP) with variables π_e, $e \in E$, for equations (1') and π_u, $u \in U$, for equations (2').

$$\max \sum_{e \in E} \pi_e + \sum_{u \in U} \pi_u \tag{DRLP}$$

subject to

$$\sum_{e \in E} a_{Re} \pi_e + \pi_u \leq c_R \qquad \forall R \in \widetilde{\mathcal{R}_u}, \forall u \in U; \tag{5}$$

$$\sum_{e \in E} b_{Se} \pi_e \leq c_S \qquad \forall S \in \mathcal{S}. \tag{6}$$

For a unit $u \in U$ and a tour $R \in \mathcal{R}_u$ let $\bar{c}_R$ be its reduced cost

$$\bar{c}_R = c_R - \sum_{e \in E} a_{Re} \pi_e - \pi_u.$$

Then, a tour $R \in \mathcal{R}_u \setminus \widetilde{\mathcal{R}_u}$ can possibly improve the current solution of (RLP) only if $\bar{c}_R < 0$ [2].

We would like to estimate during the column generation process how far we are still away from the optimal solution of (LP). To this end, we use a bound, usually attributed to Lasdon, coming from the Langrangean relaxation of (LP) w.r.t. the constraints (1).

Lemma 3.1. *Let (π_e^*, π_u^*) be an optimal solution to (DRLP) and $(x_R^*, x_S^*)^T$ be the corresponding primal solution. Then the cost c_{LP}^{opt} of an optimal solution of (LP) satisfies*

$$c_{LP}^{opt} \geq \sum_{R \in \widetilde{\mathcal{R}}} c_R x_R^* + \sum_{S \in \mathcal{S}} c_S x_S^* + \sum_{u \in U} \min_{R \in \mathcal{R}_u} \left(c_R - \sum_{e \in E} a_{Re} \pi_e^* - \pi_u^* \right) \tag{7}$$

□

This lower bound is useful in the course of a column generation algorithm since its main terms have to be computed during the column generation process anyway.

4. The Algorithm

In the following we outline the algorithm in a top-down manner. In order to make it readable we refrain from introducing symbols for parameters used to guide the algorithm. We also do not use formulas to describe it. We rather present a legible version that we find is easier to decipher than a rigorous mathematical formulation.

Three key ingredients are designed to ensure fast convergence of the reduced LP solutions:

(1) pricing on a initially small but dynamically growing search space,
(2) acceptance of columns on the basis of a dynamically updated acceptance threshold,
(3) variation in the sorting criterion influencing the selection of the search subspace in the branch&bound pricing algorithm (in the default setting we alternatingly sort by reduced costs, completion time, and primal costs).

For further reference, we call the concert of these methods *Dynamic Pricing Control*.

```
    Input: Instance of VDP
    Output: optimum dispatch
  initialize LP, ℓ, d, a, L, t, <
  while true do
    repeat
      {generate new columns in search tree of degree d and depth ℓ:}
 5:   AddNewColumns((LP, u, ℓ, d, a, <))
      double [halve] a if less [more] than 1000 columns were generated
      solve LP
      update dual prices
      if LP progress sufficient and elapsed time large enough then
10:     solve IP corresponding to LP with time limit t
        increase t
        output corresponding dispatch to a file
      end if
      if optimality check successful then
15:     mark LP as optimal
        break
      end if
      increase ℓ
    until LP progress stalls and ℓ > L
20:  if LP marked optimal then
      break
    end if
    increase d
    set ℓ to initial value
25: end while
  solve IP corresponding to LP to optimality
  return corresponding dispatch
```

Algorithm 4.1: ZIBDIP

4.1. **Top Level Algorithm.** The input of the top level algorithm in ZIBDIP is an instance of the VDP.

The initial LP consists of all elementary tours for all contractors plus a tour for each unit from its current position to its home position (*go-home-tour*). This way, both the initial LP and the initial IP are feasible.

The search for additional columns is done in a *depth-first-search branch&bound tree* (*search tree*, for short) for each unit. Each node in the search tree corresponds to a tour starting at the current position of a unit and ending at the position of the last event served by the unit. A node can be completed to a feasible tour by appending the tour from the position of the last event in the node to the unit's home position. The *pre-cost* of a node in the search tree is the *reduced cost* [2, 12] of the corresponding tour (without returning to the unit's home position and overtime). The *cost* of a node in the search tree is defined as the reduced cost of the corresponding feasible tour (including the costs for returning to the home position and overtime). The *dual prices* for the events and units are taken from the previous run of the LP solver.

Input: Linear program LP, maximal search depth ℓ, maximal search degree d, current node v ($v := r$ if not specified), acceptance threshold α, sorting criterion for partial tours $<$
Output: The linear program LP with additional columns
if v has length ℓ **then**
  return
end if
while less than d children visited **do**
  pick the next best child c of v according to search order $<$
  if cost of c smaller than α **then**
    add column corresponding to c to LP
  end if
  if LOWERBOUND(LP, c) smaller than α **then**
    ADDNEWCOLUMNS(LP, l, d, c, α, $<$)
  end if
end while
return LP

Algorithm 4.2: ADDNEWCOLUMNS

The root node r of the search tree corresponds to the empty tour. Given a node v in the tree, the children of v are obtained by appending one event to v that is not yet in v.

We generate columns for each unit in loops with increasing values for the maximal search depth (inner loop) and the maximal search degree (outer loop). The values for the maximal search depth are increased until no progress has been made in the previous step provided the search depth was sufficiently large, at latest when the depth equals the number of events. The search degree is increased until an optimality criterion is met or the search degree has reached the number of events (see Algorithm 4.1).

While we are adding columns to the LP we fix the upper bound to a negative acceptance threshold: all columns that have reduced costs smaller than the acceptance threshold are added to the LP. This acceptance threshold is updated after each iteration depending on the number of columns produced.

This search on a dynamically growing space ensures that

- the effort of finding new columns is small in the beginning, when the dual variables are not yet in good shape
- the dual variables are updated often in the beginning
- this update is fast since the number of columns in the LP is still small
- we can enforce the output of a feasible integer solution early
- the search is exact later in the run when the dual information is reliable

Whenever a new integral solution is found we output the corresponding dispatch.

4.2. **Column Generation.** The input of the column generation part consists of an LP (containing information on the dual prices of the previous optimum LP solution), a maximal search depth ℓ, a maximal search degree d, the current node, an acceptance threshold α, and a sorting criterion $<$ on nodes of the search tree.

Each node in the search tree of a unit is defined by an ordered sequence of events. This sequence uniquely specifies a tour for the unit. For each unit, the root node of the

Input: node v
Output: A lower bound on the cost of a node below v
for events e not in v **do**
compute the maximal gain of e below v
end for
set s to the sum of the largest $l - l(v)$ maximal gains
return pre-cost of v minus s

Algorithm 4.3: LOWERBOUND

search tree is initialized with the empty tour. The cost of the root node is the reduced cost of the go-home tour. If we have visited a node v in the search tree we recursively traverse the subtrees of at most d children of v. The children of v are constructed by appending a new event to the event sequence of v. The cost of the child is given by the reduced cost of the correnssponding feasible tour. The order in which the subtrees of the at most d children of v are traversed is given by sorting the children in the order of increasing reduced costs of the new node (*greedy*), increasing completion time of the new node (*greedy makespan*), or increasing primal cost (*primal-greedy*). (Other sorting criteria can be activated on demand.)

Whenever the subtree below a node has no chance to contain a node with cost smaller than the acceptance threshold we skip that branch (see Section 4.3). Whenever a visited node has cost smaller than the negative acceptance threshold we add it to the LP (see Algorithm 4.2).

The acceptance threshold is doubled or halved depending on how many columns were found. Since the column generation procedure is called very often, this controls the number of columns delivered. This has (in our case) certain advantages over stopping after a fixed number of generated columns (*forced early stop* [10]):

- the optimal column in the current search space is not missed
- the generated columns might cover the set of events more uniformly than in forced early stop

4.3. **Pruning the Search Tree.** The lower bound scheme for the subtree below a node v in the search tree works as follows: for all events not yet in the sequence of v, compute the *maximal gain* of serving e at some point after v as the dual price of e minus the unavoidable (primal) cost of serving e no earlier then the completion time of the last event in v. This unavoidable cost is the late cost plus the overtime cost incurred when e were served next. If v has length $l(v)$ then we call the sum of the largest $l - l(v)$ maximal gains the *total maximal gain below* v). The pre-cost of v minus the total maximal gain below v is a lower bound for the cost of the cheapest node below v (see Algorithm 4.3). This lower bound estimation is fast and effective for our type of input data.

4.4. **The start heuristic.** We considered the alternative of starting the column generation process from a "good" initial solution constructed heuristically. The heuristic we used consists of two phases. In the first phase, a dispatch is greedly built using a best insertion approach. We start with all units having "return-home" tours (i.e., tours without events) assigned to them. Then we consider all events sorted by increasing

deadlines, which is the order in which they are given in our data sets. For each event e, we search for the best unit's tour—and the position within this tour—where e can be inserted. If the cost for inserting e at this position is smaller than the cost of assigning e to the best contractor capable of servicing it, the insertion is done and we continue with the next event.

Input: Instance of VDP
Output: valid dispatch
{Phase I: construct dispatch greedily by best insertion:}
initialize R_u as "return-home" tour $\forall u \in U$
for $i \in \{1, \ldots, |E|\}$ **do**
  determine route R_u^* and position k^* for which the cost $\Delta c(R_u^*, k^*)$ of inserting e_i is minimal
  if $\Delta c(R_u^*, k^*)$ is less than the cost of assigning i to its best contractor **then**
    insert e_i in R_u^* at position k^*
  end if
end for
{Phase II: improve dispatch by node-exchanges while possible:}
while $\exists i, j \in E$ with XCHCOST$(e_i, e_j) < 0$ **do**
  execute node-exchange between events e_i and e_j
end while

Algorithm 4.4: STARTHEURISTIC

Input: two events e_i, e_j
Output: amount of increment in the dispatch cost if a node exchange between the events e_i and e_j is executed
if e_i and e_j are currently assigned two different (unit or contractor) tours **then**
  Let $T_1 = (u|v, e_{i_1}, \ldots, e_{i_k}, e_i, e_{i_{k+1}}, \ldots, e_{i_n})$
  and $T_2 = (u|v, e_{j_1}, \ldots, e_{j_l}, e_j, e_{j_{l+1}}, \ldots, e_{j_m})$
  be the old tours
  compute the costs $c_{T_1^*}$ and $c_{T_2^*}$ of the new tours
  $T_1^* := (u|v, e_{i_1}, \ldots, e_{i_k}, e_j, e_{i_{k+1}}, \ldots, e_{i_n})$
  $T_2^* := (u|v, e_{j_1}, \ldots, e_{j_l}, e_i, e_{j_{l+1}}, \ldots, e_{j_m})$
  $\Delta := c_{T_1^*} + c_{T_2^*} - c_{T_1} - c_{T_2}$
end if
if both e_i and e_j are currently assigned to the same (unit) tour **then**
  Let $T = (u, e_{k_1}, \ldots, e_{k_n})$ with $i = k_r$ and $j = k_s$
  be the old tour. Assume w.l.o.g. $r < s$.
  compute the cost of the new tour
  $T^* := (u, e_{k_1}, \ldots, e_{k_{r-1}}, e_{k_s}, e_{k_{r+1}}, \ldots, e_{k_{s-1}}, e_{k_r}, e_{k_{s+1}}, \ldots, e_{k_n})$
  $\Delta := c_T^* - c_T$
end if
return Δ

Algorithm 4.5: XCHCOST

The second phase improves the dispatch by executing some *node-exchange* steps. We define a node-exchange as an interchange between the current positions of two events e_i and e_j. We considered hereby three possible cases: both e_i and e_j might have been assigned to the same unit's tour, to different tours, or one of them might currently be served by a contractor. In all cases, the step consists in replacing e_i by e_j in the node sequence that describes the route currently covering e_i and viceversa. Exchanges are taken whenever they lead to improvements of the solution. The heuristic stops when no more improving steps can be made (see Algorithm 4.4).

To reduce the running-time of the heuristic, we used certain rules for discarding insertions/node-exchanges that had no chance (or very little chance) of leading to an improvement in the objective function. They are all based upon one simple idea which takes advantage of the special structure of the cost function in our problems. For any event e, it is easy to derive from its soft time window a "hard" deadline θ_e^h after which the lateness cost $c_e^{\text{late}}(\theta_e^h - \theta_e^d)$ equals or exceeds the cost of assigning e to the cheapest contractor capable of servicing it. Since this deadline has to be respected in any optimal solution, all units that can't reach e before that time are not considered for insertions/exchanges. In all test datasets, it turned out that this hard deadline was narrow enough to reduce the number of "reasonable moves" drastically.

The heuristic obtained a solution in less than an half second for ten of the twelve instances considered. (The other two demanded about 1.3s.) The gap between the solution value and the LP-lower bound LB (see section 5.2.3) varied between 2.1% and 5.3% for the instances belonging to the "low" and "medium" load configurations, and between 7.3% and 19.2% for the "high" and "extreme" load cases. Although these values are in general significantly better than the first ones found by the column generation algorithm alone (i.e., without using start heuristic), it turned out in all cases that this advantage was rapidily consumed during the first 5 seconds. The column generation process seems thus to be very efficient in finding all good tours constructed by the heuristic.

4.5. **Default Settings.** We used the following start values as defaults in ZIBDIP. The values were not changed during the test runs.

- The initial LP contains all elementary contractor tours plus the tours coming from a column generation step with degree 1 and unbounded depth where each event was assigned to its closest unit in advance.
- The initial search depth is set to $\ell := 3$.
- The initial search degree is set to $d := 1$.
- The initial acceptance threshold is set to $a := -1$.
- The minimal tour length that is searched for is set to $L := |E|/|U| + 3$.
- The time bound for solving the reduced IP is set to 10% of the elapsed computing time.
- The initial order $<$ is set to reduced cost sorting, followed by completion time sorting and primal cost sorting.

4.6. **Limitations.** The efficiency of ZIBDIP depends on the relation of late costs to drive costs. Rule of thumb: the larger the late costs are in comparison to the drive costs the better is the performance of ZIBDIP and vice versa. Also, whenever tours of length

more than ten events are in an optimal solution, our method significantly degrades in performance.

Consequently, the algorithm is not suitable for dispatching without effective time windows. It is also not suitable when the drive costs dominate the late costs.

In the case of the particular dispatching problem of ADAC, the mentioned limitations do not apply.

4.7. **Remarks on the Test Implementation.** The algorithm was implemented in standard C++ using the GNU C++ compiler version 2.95.2 [4]. Standard container structures were taken from the Standard Template Library (`vector`, `list`, `priority queue`) [9]. For the computation of distances a distance cache was employed.

5. Computational Results on Real-World Data I: Our Model versus a Genetic Algorithm Approach

In this section, we present computational results and a comparison to an experimental prototype code AD2 which has been kindly provided by Intergraph Public Safety (IPS), our industrial partner in addition to the ADAC. The key learning of this section is that for the type of data in the real-world ADAC problem the method based on Integer Programming outperforms a professional code based on primal heuristics and genetic algorithms.

5.1. **Prototype Codes.** The first prototype is an experimental code AD2.exe, which is based on a genetic algorithm combined with a number of heuristics. AD2 was carefully produced by our industrial partner in order to make the most out of the genetic algorithms approach. It was evaluated in four parameter-settings proposed by the producer (see Section 5.2.2).

The code ZIBDIP was run with and without the start heuristic ZHEU described in Section 4.4. Other than that, the same configuration was used on all test instances: the initial search depth was set to 3, the initial search degree was set to 1. The initial acceptance threshold was set to 1.

The code AD2 is based on a genetic algorithm approach GA which can be initialized with two different start-heuristics IT and CL. An additional best-insertion type heuristic HEU may also be used in the solution process. This prototype was evaluated in four parameter-settings proposed by the producer. The code ZIBDIP was run with the same configuration on all test instances: the initial search depth was set to 3, the initial search degree was set to 1. The initial acceptance threshold was set to -1.

5.2. **Test Setup.** In this section we explain the setup which was used for the tests. The evaluation followed the scientific guidelines proposed in [7, 8]. Section 5.2.1 describes the test data, Section 5.2.2 lists the hardware and software setup and Section 5.2.3 specifies the evaluation guidelines given by the ADAC.

5.2.1. *Test Data.* The test data was provided by ADAC/IPS from the database archive of the currently running PANDA-system. For each of four "system load situations", low, medium, high and extreme, a number of data sets were selected. The test sets vary in the number of open events (# events) which must be serviced and the number of ADAC-units (# units). The three extreme load data sets were obtained by duplicating events in smaller data sets. All other data sets consisted purely of snapshots of the

Data File	System Load	# events	# units
1403_low	low	106	84
1903_low	low	44	73
2702_low	low	93	74
0904_medium	medium	104	96
1504_medium	medium	125	93
1602_medium	medium	119	94
1704_high	high	156	83
1903_high	high	196	108
2702_high	high	215	98
540e_99u	productive data, simulated extreme load	540	99
770e_200u	simulated extreme load	770	200
Xtreme	simulated extreme load	775	211

TABLE 1. Overview over the data sets used in the evaluation.

database. Table 1 lists the details of the various test instances. In all cases a single external contractor ("generic contractor") of unlimited capacity was available as an alternative serving-unit for the requests.

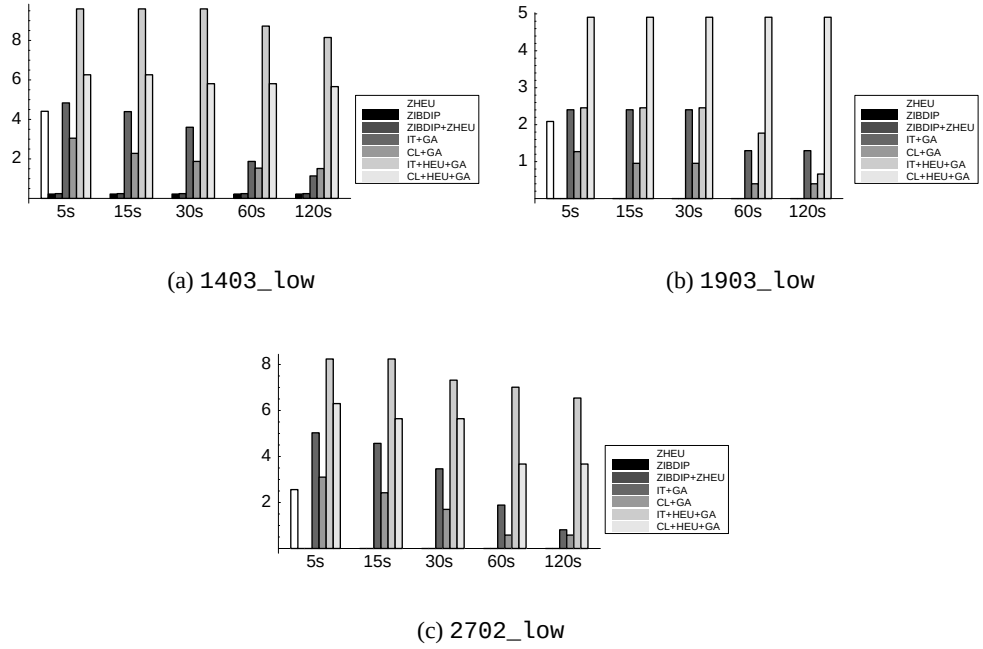
The cost factors are in the following relations (absolut values are not disclosed here): A delay of one hour is as expensive as 20 hours of driving and approximately as expensive as three contractor services. Overtime costs make for one third of the delay penalty. These data are a result of a careful internal modeling process in cooperation with ADAC and IPS.

5.2.2. *Hardware and Software Setup.* The code AD2 was used on a 933 Mhz Pentium III machine equipped with 256 MB of RAM running the Windows 2000-server operating system (version 5.00.2195, service-pack 1 installed). A single user was logged in for the total evaluation period. As suggested by the producer, the prototype was run with four different settings: iterated combined with genetic algorithm (IT+GA), close combined with genetic algorithm (CL+GA), iterated combined with heuristics and genetic algorithm (IT+HEU+GA) and close combined with heuristics and genetic algorithm (CL+HEU+GA).

The ZIBDIP-prototype was used on a 800 Mhz Pentium III machine equipped with 256 MB of RAM running Linux as operating system (kernel version 2.2.16). As in the case of AD2, a single user was logged in during the test. The prototype ZIBDIP was compiled using the GNU C++-compiler gcc, version 2.95.2 19991024 (release) [4]. The compiler flags were `-O6 -mpentiumpro` (`-O6`: optimize, `-mpentiumpro`: produce code for Pentium II and later processors). ZIBDIP used the linear programming (LP) and integer linear programming (ILP) solver CPLEX version 7.0 [6]).

The ZIB-prototype was run in two combinations, the core-algorithm ZIBDIP based solely on dynamic column generation and a combination ZIBDIP+ZHEU of this approach together with the start heuristic from Section 4.4.

Virtual memory effects were not an issue during the test series. All codes ran in RAM without swapping.



(a) `1403_low` (b) `1903_low`

(c) `2702_low`

FIGURE 2. Results for low system load.

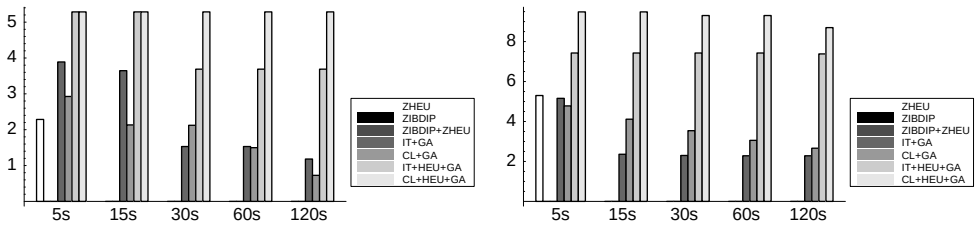
5.2.3. *Evaluation Guidelines.* Both prototypes were used with the described parameter settings on all test instances. The solutions computed by the prototypes after 5, 15, 30, 60 and 120 seconds were evaluated. For the largest test instances `540e_99u`, `770e_200u` and `Xtreme` we also recorded the solution quality delivered after 600 seconds. The measure used to judge the quality of the solutions we used the *relative error* in percent with respect to the LP-lower bound LB for the optimal solution.

The LP-lower-bound LB was used instead of the "real" (integral) optimal solution, since computing a proof of the optimality of the currently-at-hand integral solution turned out to be time consuming. Moreover, for all test instances the optimal (integral) solution was at most 0.494% above the lower bound and for most instances actually coincided with the lower bound; hence the relative error with respect to the lower bound LB yields essentially the same results as the comparison with the optimal (integral) solution.

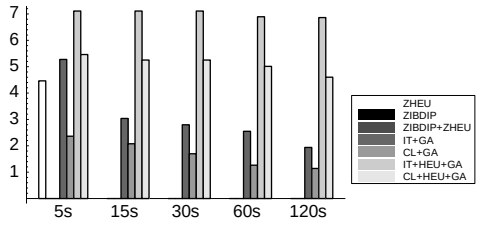
5.3. **Results.** This section is dedicated to the presentation of the computational results of the two prototypes on the 12 test instances described in Table 1.

5.3.1. *Low Load.* Three test instances representing "low load" situations were fed into the two prototypes: `1403_low`, `1903_low` and `2702_low`. The results are shown in Figure 2. The solution quality of the start heuristic ZHEU which terminated in less than 0.01 seconds for each of the instances is shown at the 5 second mark.

ZIBDIP found provably optimal solutions for all but the first instance (`1403_low`) within the first 5 seconds of computing (in fact, all these solutions were determined within less than five seconds of computing time). For `1403_low` a solution with



(a) `0904_medium` (b) `1504_medium`



(c) `1602_medium`

FIGURE 3. Results for medium system load.

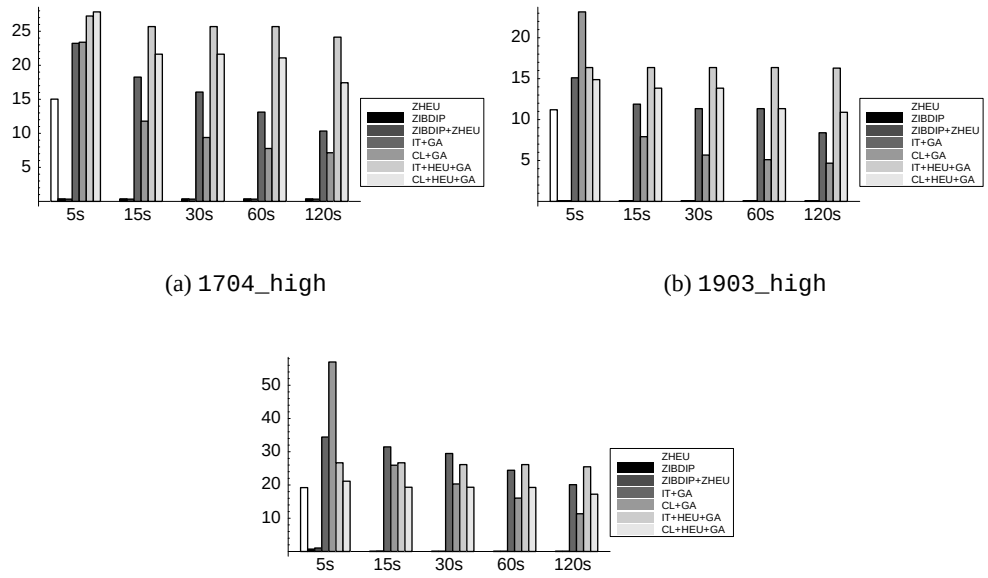
0.22% error was output within the 5 second time limit (after 0.45 seconds). The use of the start heuristic (ZIBDIP+ZHEU) appeared to be unnecessary in all cases.

The best parameter setting for AD2 was CL+GA. Within the critical 5 second time limit solutions with relative error of at most 5% could be determined. IT+GA performed similarly. The combinations involving the heuristics, IT+HEU+GA and IT+CL+GA, initially found solutions with relative error of about 3–9% which were not improved substantially until the 60 second time limit.

5.3.2. *Medium Load.* The test data for "medium load" consisted of the three instances `0904_medium`, `1504_medium` and `1602_medium`. The results obtained are shown in Figure 3.

All algorithms behaved similarly to the "low load" situation. ZIBDIP found provably optimal solutions for all instances within the first 5 seconds of computing (optimality certificates were obtained after 0.48 s, 0.70 s and 0.43 s). The combination involving the start heuristic (ZIBDIP+ZHEU) performed the same. The start heuristic ZHEU consumed less than 0.02 seconds of computing time on each of the three instances.

As in the "low load" cases the best parameter setting for AD2 was CL+GA which delivered solutions with maximum error of 5% within the first 5 seconds of computing. IT+GA was slightly worse and failed to beat the 5%-error barrier within the first 5 seconds twice, albeit only by a marginal amount.



(a) `1704_high` (b) `1903_high`

(c) `2702_high`

FIGURE 4. Results for high system load.

5.3.3. *High Load.* To determine the behavior of the prototypes in "high load" situations the three data sets `1704_high`, `1903_high` and `2702_high` were given to the prototype codes. Figure 4 displays the results.

ZIBDIP found solutions with maximum error 0.7% within the first 5 seconds of computing time, independent from whether the start heuristic was used or not. The start heuristic ZHEU needed at most 0.05 seconds to terminate for instances `1704_high` and `1903_high`, and 0.13 seconds for `2702_high`.

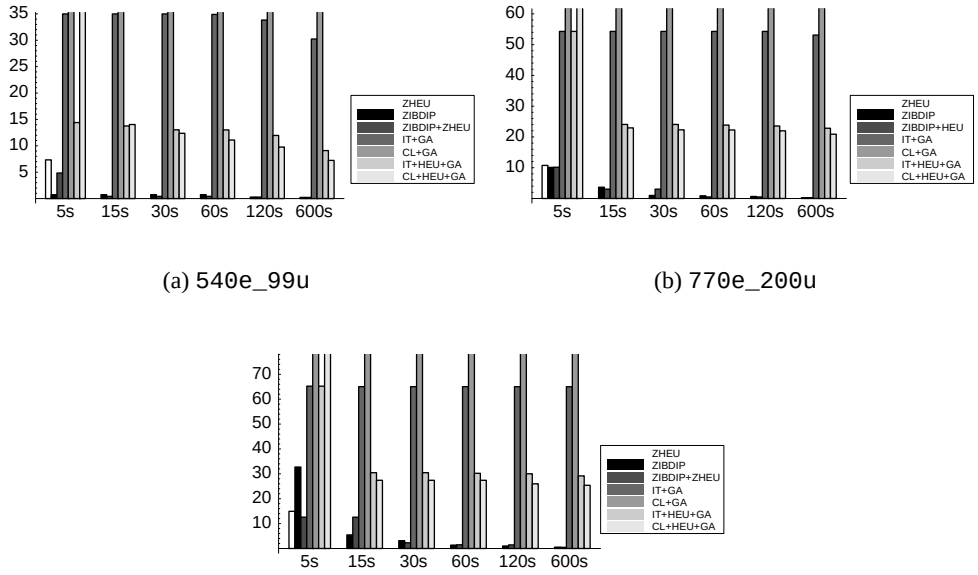
Instance `2702_high` turned out to be the most difficult to handle for the prototypes. ZIBDIP provided a 0.70% error after 4 seconds.

AD2 in its best setting for this data set (CL+HEU+GA) could not get below an error of 20%.

In general, AD2 showed some degradation in performance on the data sets. For none of the instances a solution with error better than 15% could be computed within the first 5 seconds. To achieve an error of less than 10% it was necessary to run the prototype for 30–60 seconds.

5.3.4. *Extreme Load.* Three test instances representing "extreme load" situations were used to test the two prototypes: `540e_99u`, `770e_200u` and `Xtreme`. Due to the size of the instances in addition to the previously listed time bounds the output of the prototype after 600 seconds of computing were logged. The computational results are shown in Figure 5.

The trends observed for the "high load" instances continued for the "extreme load" data. The ZIB-prototype ZIBDIP degraded slightly in performance on the "extreme load" data sets compared to the smaller examples from the previous sections. Instance



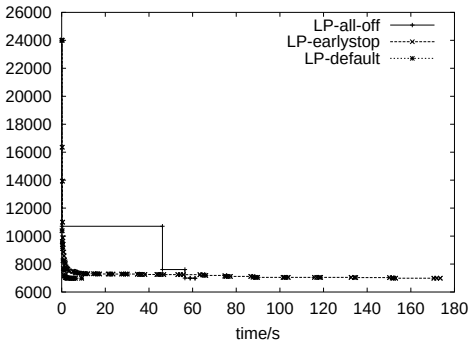
(a) `540e_99u` (b) `770e_200u`

(c) `Xtreme`

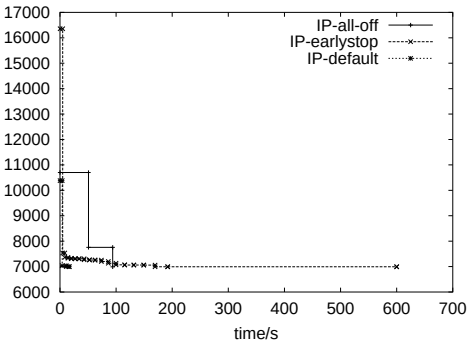
FIGURE 5. Results for extreme system load.

`540e_99u` was easiest to handle by ZIBDIP and a solution with 1.0% error was available after 5 seconds. This solution could be improved to a 0.28% error after two minutes and to 0.21% error after the maximum of 600 seconds. For `770e_200u` ZIBDIP started with 10.1% error after 5 seconds (which is still roughly a factor of 5 better than the solution found by the best AD2 setting and roughly 40 times as small as the error of CL+GA which was the best AD2 setting for smaller instances). The last instance, Xtreme, appeared to be the most difficult for ZIBDIP to handle. After 5 seconds the error was 33% (compared to 65% for the best setting in AD2), after 15 seconds an error of only 5.5% remained. Still, even in this case the optimality gap could be decreased to 0.53% before the 600 second time bound. For instance `Xtreme` the use of the start heuristic (ZIBDIP+ZHEU) was advantageous. In combination with the heuristic the benchmark error after 5 seconds of computing time was only 13% compared to 33% without the heuristic. However, for the other "extreme load" data sets the start heuristic did not help to improve the results. The start heuristic delivered its solution on each instance in less than 1.3 seconds.

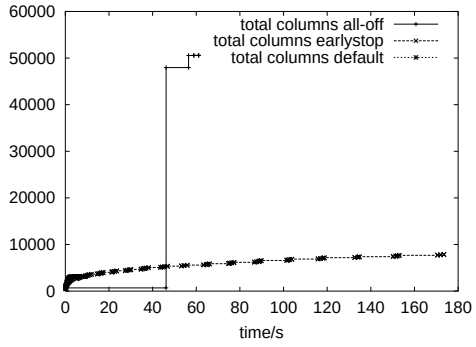
AD2 took a serious dip in performance. After 5 seconds only in one case (setting IT+HEU+GA applied to `540e_99u`) a solution with error less than 20% could be determined. In all other cases the error was larger than 50%. Even after 600 seconds of computing the best solutions could not beat the 5% error barrier. The setting CL+GA more or less broke down completely. For none of the data sets solutions with less than 400% error could be found within the first 5 seconds of computing time, after 600 seconds still about 100% error remained in the best case.



(a) RLP values



(b) RIP values



(c) Columns in RLP

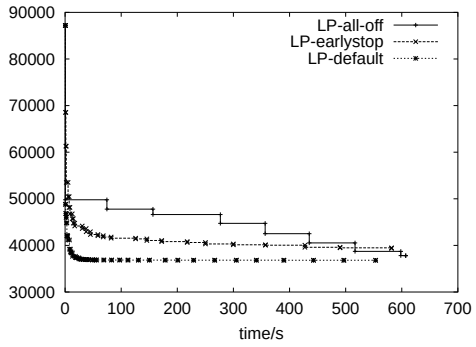
FIGURE 6. Results for `2702_high`

6. COMPUTATIONAL RESULTS ON REAL-WORLD DATA II: EFFECT OF DYNAMIC PRICING CONTROL

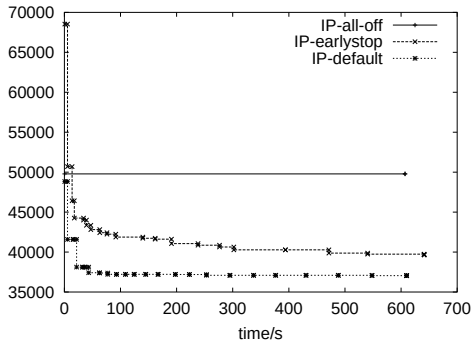
While in the last section we have shown that the model and the algorithmic approach are more suitable for our problem than an approach based on primal heuristics and genetic algorithms, we add some more information on the performance of algorithmic variants of ZIBDIP.

In particular, in order to show the effectiveness of the ZIBDIP algorithm, we compared (on the basis of one high and one extreme load data set from the previous section) the following setups:

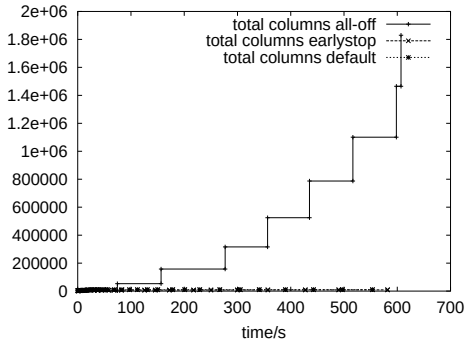
- an *unmodified* column generation procedure, where the pricing step is done on the whole search space and all columns with negative reduced costs are included in the new reduced LP (all-off)
- *forced early stop* in the pricing step: whenever one column with negative reduced costs is found then the pricing step is interrupted, and the new column is added to the reduced LP (this method was successfully applied by [10] on modifi ed instances of some Solomon problems (earlystop),



(a) RLP values



(b) RIP values



(c) Columns in RLP

FIGURE 7. Results for `Xtreme`

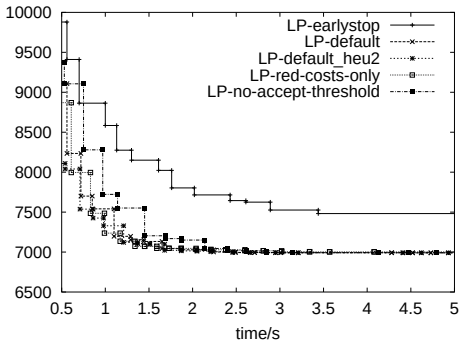
- the default settings of ZIBDIP (default).

We have plotted over time the development of the optimal solutions of the reduced LPs (RLP) and the corresponding IPs (RIP) as well as the accumulated number of columns.

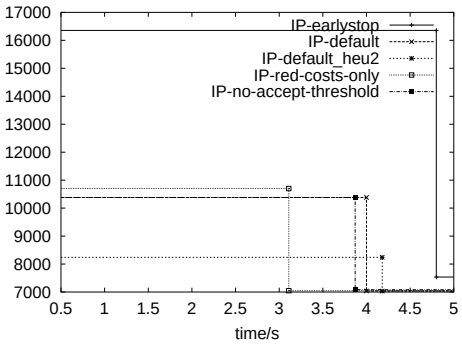
The results in Figures 6 and 7 show that a vanilla column generation algorithm is not capable to meet the real time requirements: far too many generated columns lead to an unacceptable solution time both in pricing and in LP solving. This, by the way, does not change if the column generation step is interrupted after the optimal column was found. It can be seen that *forced early stop* is for our special kind of problem not the proper approach to deal with this diffculty: too many iterations are wasted by adding columns with inferior quality, and so *forced early stop* is clearly outperformed by ZIBDIP's default settings.

Observe in Figure 6 that a successful optimality check for the default setting of ZIBDIP made the program terminate already within 15 seconds whereas the all-off and the earlystop settings need a great deal longer to nail down the optimal LP solution.

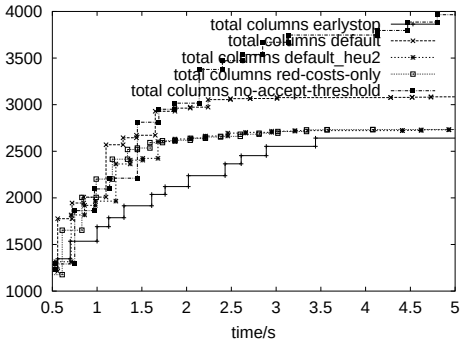
We have run this test on all of our data with the same results, except that for the low load instances the differences were not equally substancial.



(a) RLP values



(b) RIP values



(c) Columns in RLP

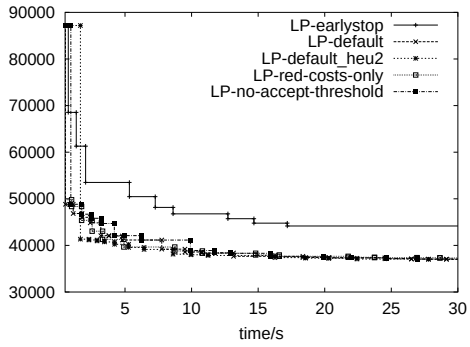
FIGURE 8. Results for `2702_high`

For a more detailed look at the effects of specific algorithmic features of ZIBDIP, we compared the performances of

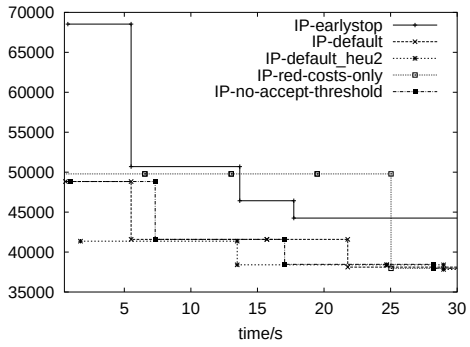
- *forced early stop* (earlystop),
- ZIBDIP in default settings (default),
- ZIBDIP with start heuristics (default_heu2),
- ZIBDIP the acceptance threshold is set to zero throughout, thus all tours with negative reduced costs found in the restricted searchspace are accepted (no-accept-threshold),
- ZIBDIP without the variation of sorting criteria in the pricing step: tours are always extended in the order of increasing reduced costs rather than according to alternating sorting criteria (red-costs-only).

We concentrate on the early phase that is most important for the realtime-application (5 seconds for high load instances, 15 seconds for extreme load instances). Figures 8 through 10 show the performance indicated by the objective values and the number of generated columns.

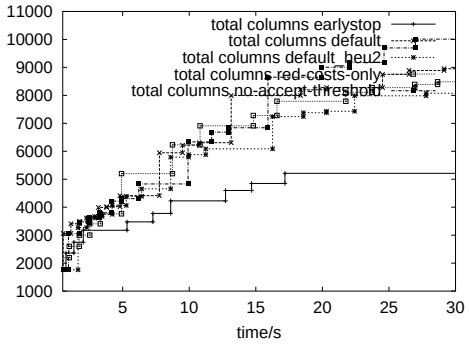
We add one additional set of input data `prob700` with 700 events and 100 units. It was created randomly and has release times in the future. It serves as a stability



(a) RLP values



(b) RIP values



(c) Columns in RLP

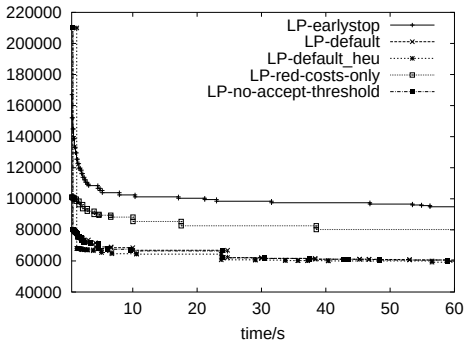
FIGURE 9. Results for `Xtreme`

checker for our methods since we expect that its uniform structure and relatively small late penalties allow for longer tours and make it harder for ZIBDIP to find the optimal tours.

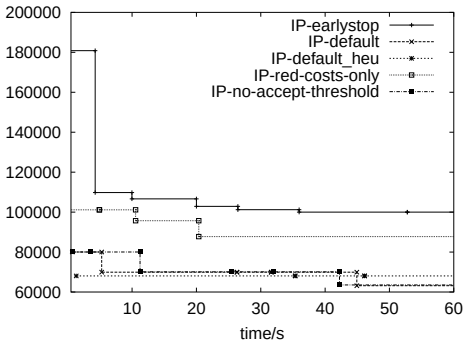
The results in Figures 8 through 10 show that using the starting heuristics ususally improves the IP solutions of ZIBDIP during the first five seconds. The convergence of the further column generation procedure is, however, not affected by the start heuristics.

Dropping the dynamic acceptance threshold leads to more columns resulting in a slightly worse performance in the speed of LP convergence. This behaviour is, however, by far not as significant as the restriction of the search space: once the search space is restricted acceptance control of new columns is a fine tuning issue.

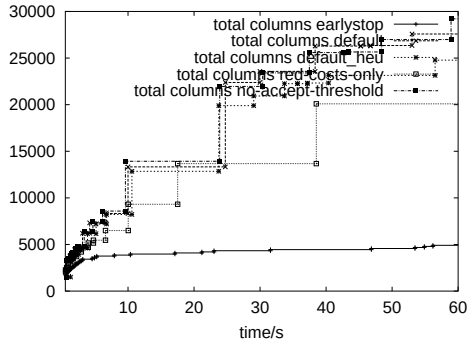
The influence of the variation of sorting criteria, while neglectable on the real-world data, is strong in the random data example. This is plausible because a greedy search by reduced costs is more promising in the absence of long tours. Since `prob700` has tours of length 9 in its near-optimal solution while for `Xtreme` there are only tours of length 4, a pure greedy search is to "narrow-minded" to find the best tours early.



(a) RLP values



(b) RIP values



(c) Columns in RLP

FIGURE 10. Results for `prob700`

Anyway: since varying the sorting criterion does not harm in the other test cases it seems advisable to use it in order to be weaponed against pathologic input data.

7. CONCLUSION

We have presented the specialized column generation algorithm ZIBDIP that solves a real-world large scale vehicle dispatching problem with soft time windows under realtime requirements. The problem arises as a subproblem in an online-dispatching task that was proposed to us by the German Automobile Association (ADAC). The algorithm clearly outperforms an experimental prototype code based on primal heuristics and genetic algorithms provided by our industrial partner. Moreover, ZIBDIP is able to provide a lower bound based on an optimal LP solution in seconds for all real-world instances provided by ADAC.

It was shown that the concept of Dynamic Pricing Control can signifi cantly speed up convergence of the column generation process, thereby making a method that has proven to be effective for large scale offline problems ready for the use in online-algorithms under realtime requirements. The most important ingredient is the dynamically growing search space, whereas the dynamic acceptance threshold and the

variation of sorting criteria rather help to make the algorithm robust against pathologic input data.

Another interesting experience is that of-the-shelf MIP solving by CPLEX was good enough to provide us with near-optimal intergral solutions. Nevertheless, in extreme load situations the time to repeatedly solve IPs can probably be reduced signifi cantly by employing clever rounding schemes. Since these are rare cases in practice, this was not considered necessary by our project partners.

The practical impact of this work is that ZIBDIP is being reimplemented into the new commercial standard automatic dispatch system distributed by IPS, one of the main providers of dispatching software, superseding the former code based on primal meta-heuristics. Moreover: this product will fi nally be used in ADAC's help centers.

8. ACKNOWLEDGEMENTS

We thank the ADAC and IPS for their cooperation, in particular for providing us with the production data and the permission to publish the computational results. Moreover, we thank IPS for their program code AD2 that was used as an adversary against ZIBDIP in this paper. We are indepted to Ralf Borndörfer for many helpful discussions.

REFERENCES

1. Julien Bramel and David Simchi-Levi, *On the effectiveness of set covering formulations for the vehicle routing problem with time windows*, Operations Research **45** (1997), no. 2, 295–301.
2. Vasek Chvatal, *Linear programming*, Freeman, New York, 1983.
3. Jaques Desrosiers, Yvan Dumas, Marius M. Solomon, and François Soumis, *Time constraint routing and scheduling*, Network Routing (Michael Ball, Tom Magnanti, Clyde Monma, and George Newhauser, eds.), Handbooks in Operations Research and Management Science, vol. 8, Elsevier, Amsterdam, 1995, pp. 35–140.
4. *GNU Compiler Collection*, Software under the GNU Public Licence (GPL), online available[3].
5. Soumia Ichoua, Michel Gendreau, and Jean-Yves Potvin, *Diversion issues in real-time vehicle dispatching*, Transportation Science **34** (2000), no. 4, 426–438.
6. *ILOG-CPLEX-7.0*, Mathematical Programming Optimizer Software, Information online available[4].
7. F. Jackson, Richard H. Paul T. Boggs, Stephen G. Nash, and Susan Powell, *Guidelines for reporting results of computational experiments. report of the ad hoc committee*, Mathematical Programming **49** (1991), 413–425.
8. David S. Johnson, *A theoretician's guide to the experimental analysis of algorithms*, Preliminary partial draft available at URL: `http://www.research.att.com/~dsj/papers.html`, 2001.
9. Nicolai M. Josuttis, *The C++ standard library: a tutorial and reference*, Addison Wesley, Reading, Massachusetts, 1999.
10. Jesper Larsen, *Vehicle routing with time windows—fi nding optimal solutions effi ciently*, DORSnyt (engl.) (1999), no. 116.
11. Sushil J. Louis, Xiangying Yin, and Zhen Ya Yuan, *Multiple vehicle routing with time windows using genetic algorithms*, 1999 Congress on Evolutionary Computation (Piscataway, NJ), IEEE Service Center, 1999, pp. 1804–1808.
12. D. G. Luenberger, *Linear and nonlinear programming*, 2 ed., Addison-Wesley, 1984.
13. Éric Taillard, P. Badeau, Michel Gendreau, F. Guertin, and Jean-Yves Potvin, *A tabu search heuristic for the vehicle routing problem with soft time windows*, Transportation Science **31** (1997), 170–186.
14. Kenny Qili Zhu and Kar-Loon Ong, *A reactive method for real time dynamic vehicle routing problem*, Proceedings of the 12th ICTAI, 2000.

[3] http://www.gnu.org/gcc

[4] http://www.ilog.com/products/cplex

KONRAD-ZUSE-ZENTRUM FÜR INFORMATIONSTECHNIK BERLIN, TAKUSTR. 7, 14195 BERLIN, GERMANY
E-mail address: {name}@zib.de