IMM
INFORMATICS AND MATHEMATICAL MODELLING

Technical University of Denmark
DK-2800 Kgs. Lyngby - Denmark

2002-06-10
jla

SPEEDING UP THE SOLUTION PROCESS FOR THE VRPTW

Jesper Larsen

TECHNICAL REPORT

IMM-TR-2002-9

IMM

Speeding up the solution process for the Vehicle Routing with Time Windows using structural information

Jesper Larsen*
Department of Mathematical Modelling
Technical University of Denmark
2800 Kgs. Lyngby, Denmark

June 11, 2002

Abstract

Two ideas for using structural information for solving the Vehicle Routing Problem with Time Windows (VRPTW) is presented. The VRPTW is a generalization of the well known capacity constrained Vehicle Routing Problem (VRP). Both techniques are based on solving the VRPTW using a Branch-and-Price approach. They reveal a huge potential for using structural information when speeding up an exact approach without compromising optimality.

*E-mail: jla@imm.dtu.dk

1 Introduction

In the real world many companies are faced with problems regarding the transportation of people, goods or information. This is not restricted to the transport sector itself but also other companies for example factories have transport of parts to and from different sites of the factory, and big companies may have internal mail deliveries. As the world economy turns more and more global, transportation will become even more important in the future.

Researchers often use models exhibiting some but not all of the characteristics of real-world problems in order to test and evaluate their ideas.

In the m-TSP problem, m salesmen have to cover the cities given. Each city must be visited by exactly one salesman. All salesmen start and end their journey in the same city (called the depot). We now want to minimize the sum of the distances of the routes.

The Vehicle Routing Problem (VRP) is the m-TSP where a demand is associated with each city, and each salesman/vehicle has a certain capacity (not necessarily identical). The sum of demands on a route can not exceed the capacity of the vehicle assigned to it. Most often the objective is to minimize the sum of distances of the routes. Note that the VRP is not purely geographic since the demand may be constraining. The VRP is the basic model for a large number of vehicle routing problems.In figure 1 a typical solution to the VRP is shown. Note that the direction in which the route(s) are driven is unimportant both for the TSP and the VRP.

By adding a *time window* to each customer in the VRP the VRPTW is obtained. In addition to the capacity constraint, the vehicles also have to adhere to the time windows of the customers. A vehicle may arrive before the time window "opens", but the customer can not be serviced until the time window "opens". It is not allowed to arrive after the time window has "closed". Some models allow for early or late servicing but with some form of penalty. These models are denoted "soft" time window models. By far the most research has been made on "hard" time window models. In this paper we also assume the time windows are "hard".

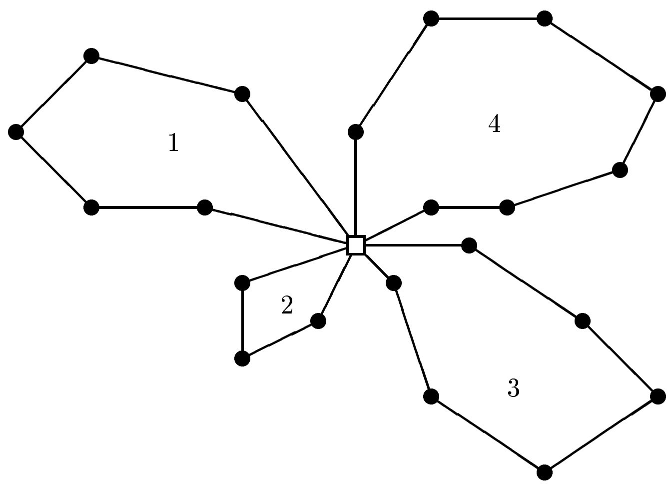

Figure 1: A typical solution to a VRP instance (4 routes). The square denotes the depot.

In section 2 a mathematical model of the VRPTW is presented. Section 3 describes the column-generation approach to solving the VRPTW. In section 4 and 5 two ways of using additional information in order to speed up the execution time are presented, and finally the paper finishes with a conclusion.

2 A mathematical model of the VRPTW

The VRPTW is given by a fleet of homogeneous vehicles (denoted $\mathcal{V}$), a set of customers $\mathcal{C}$ and a directed graph $\mathcal{G}$. The graph consists of $|\mathcal{C}|+2$ vertices, where the customers are denoted $1, 2, \ldots, n$ and the depot is represented by the vertices 0 ("the driving-out depot") and $n+1$ ("the returning depot"). The set of vertices, that is, $0, 1, \ldots, n+1$ is denoted $\mathcal{N}$. The set of arcs (denoted $\mathcal{A}$) represents connections between the depot and the customers and among the customers. No arc terminates in vertex 0, and no arc originates from vertex $n+1$. With each arc (i, j), where $i \neq j$, we associate a *cost* c_{ij} and a *time* t_{ij}, which may include service time at customer i.

Each vehicle has a capacity q and each customer i a demand d_i. Each customer i has a *time window* $[a_i, b_i]$. A vehicle must arrive at the customer before b_i. It can arrive before a_i but the customer will not be serviced before. The depot also has a time window $[a_0, b_0]$ (the time windows for both depots are assumed to be identical). $[a_0, b_0]$ is denoted the *scheduling horizon*. Vehicles may not leave the depot before a_0 and must be back before or at time b_{n+1}.

It is assumed that q, a_i, b_i, d_i, c_{ij} are non-negative integers, while the t_{ij}'s are assumed to be positive integers. It is also assumed that the triangular inequality is satisfied for both the c_{ij}'s and the t_{ij}'s, that is, for vertices $i, j, l \in \mathcal{N}$ $c_{ij} + c_{jl} \geq c_{il}$ and $t_{ij} + t_{jl} \geq t_{il}$.

The model contains two sets of decision variables x and s. For each arc (i, j), where $i \neq j, i \neq n+1, j \neq 0$, and each vehicle k we define x_{ijk} as

$$x_{ijk} = \begin{cases} 0 & \text{if vehicle } k \text{ does not drive from vertex } i \text{ to vertex } j \\ 1 & \text{if vehicle } k \text{ drives from vertex } i \text{ to vertex } j \end{cases}$$

The decision variable s_{ik} is defined for each vertex i and each vehicle k and denotes the time vehicle k starts to service customer i. In case the given vehicle k does not service customer i s_{ik} does not mean anything. We assume $a_0 = 0$ and therefore $s_{0k} = 0$, for all k.

We want to design a set of minimal cost routes, one for each vehicle, such that

- each customer is serviced exactly one time,
- every route originates at vertex 0 and ends at vertex $n+1$, and
- the time windows and capacity constraints are observed.

The VRPTW can then be stated mathematically as:

$$
\begin{aligned}
&\min \sum_{k\in\mathcal{V}} \sum_{i\in\mathcal{N}} \sum_{j\in\mathcal{N}} c_{ij} x_{ijk} \quad s.t. && && (1)\\
&\sum_{k\in\mathcal{V}} \sum_{j\in\mathcal{N}} x_{ijk} &=&\ 1 & \forall i \in \mathcal{C} \quad & (2)\\
&\sum_{i\in\mathcal{C}} d_i \sum_{j\in\mathcal{N}} x_{ijk} &\leq&\ q & \forall k \in \mathcal{V} \quad & (3)\\
&\sum_{j\in\mathcal{N}} x_{0jk} &=&\ 1 & \forall k \in \mathcal{V} \quad & (4)\\
&\sum_{i\in\mathcal{N}} x_{ihk} - \sum_{j\in\mathcal{N}} x_{hjk} &=&\ 0 & \forall h \in \mathcal{C}, \forall k \in \mathcal{V} \quad & (5)\\
&\sum_{i\in\mathcal{N}} x_{i,n+1,k} &=&\ 1 & \forall k \in \mathcal{V} \quad & (6)\\
&s_{ik} + t_{ij} - K(1 - x_{ijk}) &\leq&\ s_{jk} & & \\
& & & & \forall i, j \in \mathcal{N}, \forall k \in \mathcal{V} \quad & (7)\\
&a_i \leq s_{ik} \leq b_i & & & \forall i \in \mathcal{N}, \forall k \in \mathcal{V} \quad & (8)\\
&x_{ijk} \in \{0, 1\} & & & \forall i, j \in \mathcal{N}, \forall k \in \mathcal{V} \quad & (9)
\end{aligned}
$$

The constraints (2) state that each customer is serviced exactly once, and (3) state that no vehicle is loaded with more than it's capacity allows. The next three sets of equations (4), (5) and (6) ensure that each vehicle leaves the depot 0, after arriving at a customer the vehicle leaves again, and finally arrives at the depot $n+1$. The inequalities (7) state that a vehicle k can not arrive at j before $s_{ik} + t_{ij}$ if it is traveling from i to j. Here K is a large scalar. Finally constraints (8) ensures that time windows are observed, and (9) are the integrality constraints. Note that an unused vehicle is modelled by driving the "empty" route $(0, n+1)$.

As mentioned earlier VRPTW is a generalization of both TSP and VRP. In case the time constraints ((7) and (8)) are not binding the problem becomes a VRP. This can be achieved by setting $a_i = 0$ and $b_i = M$ (where M is a large scalar) for all customers i.

The first paper proposing an exact algorithm for solving the VRPTW was published in 1987 in [KRT87]. The method based on column generation was used for the VRPTW for the first time in [DDS92]. In [KDM+97] 2-path cuts are added to strengthen the LP-relaxation. Implementations of algorithms for parallel computers have been presented in [CR99, LCM00].

3 Solving the VRPTW using column generation

As in many routing problems, the VRPTW can be described as a *set partitioning problem* (SPP). In this model each column corresponds to a feasible route, while the rows in the constraint matrix corresponds to customers. For each column, the variable x_r is defined by

$$x_r = \begin{cases} 1, & \text{if route } r \text{ is used in the solution} \\ 0, & \text{otherwise} \end{cases}$$

and c_r denotes the cost (distance) of route r. The resulting model becomes:

$$\min \sum_{r \in \mathcal{R}} c_r x_r \quad \text{s.t.} \tag{10}$$

$$\sum_{r \in \mathcal{R}} \delta_{ir} x_r = 1 \qquad \forall i \in \mathcal{C} \tag{11}$$

$$x_r \in \{0, 1\} \tag{12}$$

where $\mathcal{R}$ is the set of **all** feasible routes, and δ_{ir} is 1 if customer i is serviced by route r and 0 otherwise.

Note that the constraints concerning time windows, capacity and flow are assumed to be respected as $\mathcal{R}$ only contains feasible routes. This makes the model very versatile as other demands can be hidden from the set partitioning formulation by the "route generator". Note also that a solution to the set partitioning formulation does not completely state a solution to the VRPTW as the order of service of the customers in each route is not given. This information has to be supplied separately, that is, two representations of the routes, one for the set partitioning problem, where we just indicate whether a customer is serviced on a given route or not, and one where the actual lay-out of the route is given, have to be maintained.

The (SPP) formulation contains two problems: first the number of feasible columns is very large even for problems with only a small number of customers, and secondly the set-partitioning problem is $\mathcal{NP}$-hard and therefore very difficult to solve.

Therefore we relax the integrality constraints (12) and use column generation to generate the columns needed.

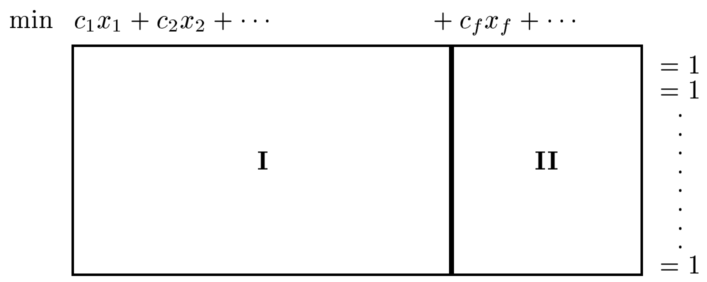

Figure 2: An illustration of the column generation approach. The columns in part I are physically present, whereas the columns in part II are "represented" by the pricing algorithm.

In figure 2 part I of the constraint matrix are the routes actually generated and included while the remaining routes (part II) are "represented" by a *pricing algorithm*. If no variables have negative reduced costs then the current optimal solution can not be improved further, one normally says that "all variables price out correctly". In the case that the variables do not price out correctly we add the column(s) with negative reduced cost to the problem, re-optimize the (SPP) and run our pricing algorithm again.

In our case, the subproblem (to be solved by the pricing algorithm) is an ESPPTWCC (see section 3.2), as all constraints are handled here. The reduced cost r_j of a non-basic variable j corresponding to a LP-solution with dual variables $\pi \in R^m$ is defined as:

$$r_j = c_j - \pi^T \delta_{\cdot j}$$

where c_j is the cost of the route and $\delta_{\cdot j}$ is the "route vector".

3.1 Branch-and-Bound

As the column generation approach does not necessarily lead to an integer solution, Branch-and-Bound is used to dynamically search through the search tree, as the original problem is divided into smaller problems.

Before starting the exploration of the search tree, we either have to produce a feasible solution for example using an approximation algorithm or a heuristic, or we may simply set the upper bound sufficiently high ("∞"), sufficiently high meaning that there exists at least one feasible solution with objective value smaller than or equal to the global upper bound. The solution is known as the *global upper bound* (or sometimes the *current best solution* or the *incumbent*). We start with the global upper bound being all *depot - i - depot* routes.

The bound used for the VRPTW is the LP relaxation of the IP model ((10)-(12)), and as a part of the bounding process, the column generation is performed.

During the execution of the Branch-and-Bound algorithm we have a set $\mathcal{U}$ of generated but unbounded descendants. The selection of which descendant should be bounded next is done by the *selection function* h, where we chose the descendant that has the lowest value. In our implementation we select the descendant having the lowest bounding function value.

Just as bounding, branching is a decision related to the problem we are trying to solve. During the years a number of different branching operations for the VRPTW has been proposed in the literature [DDS92, Hal92, Koh95, GDDS95]:

1. branching on the number of vehicles,

2. branching on the flow variables (x_{ijk}),
3. branching on the sums of flow variables, and
4. branching on resource windows.

After a number of preliminary tests we decided to implement the same branching strategy as in [Koh95]: to branch on the number of vehicles if possible and otherwise branch on flow variables.

3.2 ESPPTWCC

Using the column generation approach as introduced with the set partitioning problem as the master problem, the subproblem becomes the following mathematical model:

$$\min \sum_{i \in \mathcal{N}} \sum_{j \in \mathcal{N}} \hat{c}_{ij} x_{ij}, \qquad s.t. \tag{13}$$

$$\sum_{i \in \mathcal{C}} d_i \sum_{j \in \mathcal{N}} x_{ij} \leq q \tag{14}$$

$$\sum_{j \in \mathcal{N}} x_{0j} = 1 \tag{15}$$

$$\sum_{i \in \mathcal{N}} x_{ih} - \sum_{j \in \mathcal{N}} x_{hj} = 0 \quad \forall h \in \mathcal{C} \tag{16}$$

$$\sum_{i \in \mathcal{N}} x_{i,n+1} = 1 \tag{17}$$

$$s_i + t_{ij} - K(1 - x_{ij}) \leq s_j \forall i, j \in \mathcal{N} \tag{18}$$

$$a_i \leq s_i \leq b_i \quad \forall i \in \mathcal{N} \tag{19}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \in \mathcal{N} \tag{20}$$

Constraints (14) are the capacity constraint, constrains (18) and (19) are time constraints, while constraints (20) ensures integrality. The constraints

(15), (16) and (17) are flow constraints resulting in a path from the depot 0 and back to the depot $n+1$. The $\hat{c}_{ij}$ is the *modified cost* of using arc (i,j), where $\hat{c}_{ij} = c_{ij} - \pi_i$. Note that while c_{ij} is a non-negative integer, $\hat{c}_{ij}$ can be any real number. As we are now working with one route the index k for the vehicle has been removed.

As can be seen from the mathematical model above the subproblem is a shortest path problem with time windows and capacity constraints, where each vertex can participate at most once in the route. For this problem (sometimes denoted the Elementary Shortest Path Problem with Time Windows and Capacity Constraints (ESPPTWCC)) there is no known efficient algorithm, making the problem unsolvable for practical purposes. Therefore some of these constraints are relaxed. Cycles are allowed thereby changing the problem to the Shortest Path Problem with Time Windows and Capacity Constraints (SPPTWCC). Even though there is a possibility for negative cycles in the graph the time windows and the capacity constraints prohibits infinite cycling. Note that capacity is accumulated every time the customer is serviced.

In order to build the SPPTWCC algorithm we have to make two assumptions:

1. Time is always increasing along the arcs, i.e. $t_{ij} > 0$.
2. Time and capacity are discretized.

The algorithm maintains a set of "shortest subpaths" defined by a number of labels. A label is a state that contains a customer number, the current time t of arrival (at the given customer) and the accumulated demand d:

$$(i, t, d).$$

The cost of the label is then defined as $c(i,t,d)$. The algorithm is based on the following simple extension of the dynamic programming behind the Dijkstra algorithm:

$$\begin{aligned} c(0,0,0) &= 0 \\ c(j,t,d) &= \min_i \{\hat{c}_{ij} + c(i,t',d') \mid t' + t_{ij} = t \wedge d' + d_i = d\} \end{aligned}$$

States are treated in order of increasing time (t). Note that for each label i there may now exist more than one state. An upper bound on the number of states is given by

$$\Gamma = \sum_{i \in \mathcal{N}} (b_i - a_i)(q - 1)$$

As this is the upper limit, many of these states might not be possible, and others will not be considered as they are dominated by other states (see later).

In a straightforward implementation we maintain a set *NPS* of not processed states. Initially this set only has one member: the label $(0, 0, 0)$. As long as there exist unprocessed labels in the set the one with the lowest time is chosen and the algorithm tries to extend this to the successors of the vertex. States at vertex $n + 1$ are not processed and are therefore kept in a special set of "solutions", from which the best one is returned as the algorithm terminates. When a label has been processed it is removed from the set of unprocessed labels. The algorithm is described in pseudo-code in figure 3.

In order to make the algorithm considerably more efficient we will (like in Dijkstra's algorithm) introduce a *dominance criterion*.

Assume that for a given vertex i we have two states (i, t_1, d_1) and (i, t_2, d_2) where $c(i, t_1, d_1) \leq c(i, t_2, d_2)$, $t_1 \leq t_2$ and $d_1 \leq d_2$. Clearly as long as the extensions based on (i, t_2, d_2) are valid the extensions based on (i, t_1, d_1) are also valid, and these will always be lower in cost (or at least not higher). Therefore the label (i, t_2, d_2) can be discarded. Formally we say that (i, t_1, d_1) *dominates* (i, t_2, d_2) (or $(i, t_1, d_1) \prec (j, t_2, d_2)$) if and only if all of the following three conditions hold:

1. $c(i, t_1, d_1) \leq c(i, t_2, d_2)$.
2. $t_1 \leq t_2$.
3. $d_1 \leq d_2$.

Each time a new label is generated we have to check with the other labels

```
⟨ Initialization ⟩
NPS= {(0,0,0)}
c(0,0,0) = 0

repeat
    (i,t,d) = BestLabel(NPS)

    for j := 1 to n+1 do
        if (i ≠ j ∧ t + t_ij ≤ b_j ∧ d + d_j ≤ q) then
            ⟨ Label feasible ⟩
            if c(j, max{t + t_ij, a_j}, d + d_j) > c(i,t,d) + ĉ_ij then
                ⟨ New label better ⟩
                InsertLabel(NPS, (j, max{t + t_ij, a_j}, d + d_j))
                c(j, max{t + t_ij, a_j}, d + d_j) = c(i,t,d) + ĉ_ij

until (i = n+1)
return
```

Figure 3: The algorithm for finding the shortest path with time windows and capacity constraints. *BestLabel* returns a label with vertex different from $n+1$ and minimal accumulated time if one exists. Otherwise a label with vertex $n+1$ is returned. *InsertLabel* inserts the newly generated label in *NPS* possibly overwriting an old label if it already exists.

at the same vertex to see if the new label is dominated by some label or the new label dominates another label.

4 Reducing the number of columns

Each call to the route generator (the SPPTWCC function) typically generates more than one route. Experiments made by Kohl [Koh95] suggests that if more than one route is available they should be entered into the master problem (the relaxed (SPP)). As also suggested by other authors we impose an upper limit on the amount of columns that are entered into the master problem after each call to the route generator. In order to rank the columns, the reduced costs are used.

After a number of Branch-and-Bound nodes have been processed, the number of columns can grow very large and some of the columns might never have been part of the basis as returned by the LP-solver. Removing these columns would leave room for some new ones. Additionally this might speed up the LP-code and the work we have to do each time we start working on a new Branch-and-Bound node (as eg. checking whether any of the arcs in the path of a route are removed from the problem in the current Branch-and-Bound node). So here we have two goals: being able to handle larger problems and solving problems faster.

To check if anything can be gained, 27 problems from a testbed where used (the problems are described in [Lar99]). The program was extended with a possibility to keep track of which columns that did become member of the basis at some point. The result was that as few as 10% and as many as 28% of the columns had been in the basis some time during the execution of the program. For the bulk of the problems the percentages were around 16 to 18. So only around 1 out of 5 entered columns did ever become part of the basis.

At least from a memory point of view there is something to be gained. Whether there is something to be gained in speed is more questionable. A possible scenario is to remove the obsolete columns after every k Branch-and-Bound nodes. Todays state-of-the-art LP-solvers (as CPLEX) are very fast (our problems of more that 10000 columns are by no means large in relation to solvable LP-problems) so it is not clear whether the contribution by removing columns is significant enough to produce faster running times. A side effect is that with fewer columns in the set partitioning few columns have to be checked every time we start working on a new Branch-and-Bound node. These effects have to counterbalance the time used on removing obsolete columns and time used generating them again in case they are needed in future descendants.

To test the possibilities we ran four versions of our algorithm with column removal added. One version deleted obsolete columns after every Branch-and-Bound node, one after every 5 Branch-and-Bound nodes, one after every 10 Branch-and-Bound nodes and finally one after every 20 Branch-and-Bound nodes. In table 1 and 2 presents the running time of the algorithm not using column removal as reference and the results of deleting obsolete columns after every 20 Branch-and-Bound nodes. "SPPTWCC" indicates the number of calls to the SPPTWCC subroutine during the execution of the algorithm.

As expected removing unused columns after every Branch-and-Bound node lead to slower running times. Columns are often reused and as the algorithm is shifting from one branch of the Branch-and-Bound tree to another removing columns after every Branch-and-Bound nodes mean that columns have to be regenerated very often. It also leads to bad running times as the starting values of the dual variables are relatively bad resulting in generation of poor quality routes before better routes can be generated.

Column deletion after every 5 and 10 Branch-and-Bound nodes is in all 27 instances outperformed by column deletion after every 20 Branch-and-Bound nodes. Totally column deletion after every 20 Branch-and-Bound nodes typically is around a factor 2 better then column deletion after every 5 and 10 Branch-and-Bound nodes.

The results clearly show that running column deletion after every 20 Branch-and-Bound nodes does lead to a better algorithm. Comparing column deletion after 20 Branch-and-Bound nodes and the algorithm without column deletion reveals that only for instance 22 is column deletion after every 20 Branch-and-Bound nodes outperformed. Running without column deletion results in a running time 15% lower, but the real reason for the lower running time is that only 53 Branch-and-Bound nodes have to be checked, whereas the same number for column deletion after every 20 Branch-and-Bound nodes is 163.

It is worth noting that column deletion after 20 Branch-and-Bound nodes outperforms the code not using column deletion totally by a factor 2.5, but not because fewer Branch-and-Bound nodes have to be checked. Whereas the algorithm without column deletion has to check a total of 15702 Branch-and-Bound nodes column deletion after every 20 Branch-and-Bound nodes needs additionally around 500 Branch-and-Bound nodes (note that around

No.	Time (s)	BB	SPPTWCC
1	786.64	263	679
2	10114.34	2129	3759
3	3554.49	1057	2126
4	4906.45	1217	2453
5	240.55	95	307
6	363.02	101	325
7	605.27	107	345
8	2215.96	933	2030
9	514.46	441	983
10	788.73	465	1155
11	600.25	335	940
12	533.53	445	1069
13	910.79	381	821
14	1644.66	601	1283
15	1965.89	601	1322
16	3607.22	1091	2094
17	6550.68	1409	2688
18	1201.16	418	991
19	1396.38	479	1070
20	661.46	235	616
21	898.20	335	794
22	1525.23	53	297
23	1126.32	327	944
24	231.54	89	301
25	321.79	335	849
26	1863.87	1681	3162
27	148.35	79	339
Total	49259.23	15702	33742

Table 1: The results of hte 27 test instances not using column deletion.

No.	Time (s)	BB	SPPTWCC	Routes deleted
1	311.03	263	1367	54320
2	3011.84	2117	10712	443946
3	1237.72	1047	5413	224949
4	1287.24	1119	5731	231572
5	156.56	129	677	23867
6	167.23	101	589	23874
7	84.24	33	226	7310
8	540.80	977	5311	241709
9	124.51	333	1523	58233
10	357.83	563	2702	102475
11	233.17	337	1786	69449
12	186.13	453	2193	83094
13	382.03	379	1912	78083
14	983.68	1085	5403	212610
15	537.24	411	2094	86644
16	1203.92	935	4622	202148
17	2212.00	1557	8296	350750
18	626.24	629	3093	118743
19	531.26	547	2619	104080
20	231.11	197	933	37129
21	454.83	431	1993	77595
22	1808.42	163	1271	80373
23	658.93	315	1776	81028
24	113.05	111	558	21127
25	134.07	333	1535	55965
26	519.75	1611	9687	330128
27	87.75	75	453	13330
Total	18182.58	16251	84475	3414531

Table 2: The results of the 27 test instances using column deletion after every 20 branch-and-bound-nodes.

double as many calls to the SPPTWCC subroutine are necessary). Gains are made because the checks on columns are drasticly reduced. The effectiveness is underlined by the fact that only for instance 7 does column deletion after every 20 Branch-and-Bound nodes need significantly fewer Branch-and-Bound nodes. For instance 14 column deletion after every 20 Branch-and-Bound nodes outperforms the algorithm without column deletion with more than 50% although almost twice as many Branch-and-Bound nodes is needed.

5 Forced Early Stop

The output from our sequential algorithm for instance R203 with 25 customers of the Solomon instances normally used to test VRPTW algorithms is shown in figure 4. As the figure shows 3 routes are needed to service the 25 customers. The number in the round brackets indicates the column number of the route in the Set Partitioning formulation, while the number in the square brackets is the length of the route times 10. Hereafter follows a couple of lines of statistics of the execution.

As can be seen, the majority of the running time is spend in the root node. Note that one of the calls to the SPPTWCC functions uses over 4400 seconds. This is a major drawback considering our efforts to develop an efficient parallel code. It became essential to cut down the time used in the root node. The SPPTWCC function is based on dynamic programming, and in a effort to cut time the execution could be stopped before the underlying search tree is fully investigated. Of course the execution can only be stopped when at least one route with negative reduced cost has been generated. Otherwise the (SPP) would remain unchanged and consequently the dual variables would remain unchanged, which would result in an identical execution of the SPPTWCC function in the next call of the subroutine.

The code for the SPPTWCC function was therefore enhanced with two constants. The two constants LIMIT and MIN_COLS_PER_ITER was introduced. Already the code uses a constant called MAX_COLS_ITER which

```
---------- Solution
Problem R203 with 25 customers is solved
The solution is the following routes:

( 6648) [1533] d - 2 - 15 - 23 - 22 - 21 - 4 - 25 - 24 - 3 - 12 - d

( 6946) [1041] d - 6 - 5 - 8 - 17 - 16 - 14 - 13 - d

( 9688) [1365] d - 18 - 7 - 19 - 11 - 20 - 9 - 10 - 1 - d

---------- Statistics
This program ran on serv3 (hp9000s700).
Total execution time                      13483.83 seconds
         (Solving root 13249.63 seconds)
Time used in separation                      0.29 seconds
         Cuts generated            2
Accumulated time used in calls of SPPTWCC 13332.516 seconds
Time used in largest single SPPTWCC call  4447.04 seconds
Branching nodes examined 43 (Veh 1, Arc 20, TW 0)
        (hereof 0 where not feasible)
No of calls to SPPTW 281, Routes generated 21250
Max no of columns selected per SPPTW 200
No of multiple customers deleted explicitly 0
IP value 3914
RP value 3816.250
LP value 3798.818
-------------------------------------------------------
```

Figure 4: The result of solving R203 with 25 customers with our algorithm

is the maximum number of routes that are returned, now the variable MIN_COLS_ITER is the minimum number of routes that should be generated before we prematurely abort the generation of labels. Note that this is not necessarily the same routes that would be returned if the SPPTWCC function was run optimally and the MIN_COLS_ITER best routes was returned. The constant LIMIT is the number of labels that have to be generated before we think of prematurely stopping the SPPTWCC code. Based on the tracing from the tests mentioned earlier the values LIMIT and MIN_COLS_ITER was chosen manually.

In order to gain more knowledge on the properties of forced early stop 3 instances were selected for initial trials. R101 with 100 customers represented the easy problems as the customers in R101 has relatively small time windows and the demands limit the routes to at most 10 customers. R103 with 50 customers represented the problems that are a bit more difficult to solve, while R202 represented the really tough problems (in R202 the time windows are relatively wide and the routes can contain up to around 30 customers). For these problems the number of labels used in each execution of the SPPTWCC code was recorded. Table 3 show the results:

Problem	Cust.	SPPTWCC calls	Routes gen.	Min	Max	|Average|
R101	100	142	2098	1365	6005	2189
R103	50	178	2228	1350	20503	2573
R202	25	117	2009	422	275813	9631

Table 3: Characteristics for the “normal” trace of the 3 selected problems

All three problems have the same feature with a number of relatively time-consuming executions in the start. This is because our initial setting of the dual variables is far away from the optimal dual values. Hence a number of “heavy-duty” calls is necessary before the values are good enough to reduce the size of the underlying search tree.

The results are depicted in the tables 4 to 6.

These preliminary results were surprisingly positive. Most noteworthy is the phenomenal reduction in running time for R202 with 25 customers – from over 3300 seconds to a mere 8 seconds (more than a factor 400!).

The running time for solving the root node is the key to understanding the greatly improved performance. For R202 with 25 customers the difference between the overall running time and the time spent solving the root node is roughly around 5 seconds, which means that the time is gained entiely in solving the root node. Clearly the worse the dual variables are the more one can gain from stopping early. The drawback of stopping early is lack

R101 – 100 customers					
LIMIT	–	4000	4000	2000	1000
MIN_COLS_ITER	–	10	1	1	1
Running time (total)	19.94	16.22	16.37	16.11	72.27
Running time (root)	7.47	5.84	5.60	6.13	62.66
No. of nodes	15	15	15	15	15
SPPTWCC calls	88	89	89	117	2352
No. of routes	3970	3479	3479	3375	2555

Table 4: Testing prematurely stop of SPPTWCC on "easy" problems.

R103 – 50 customers						
LIMIT	–	10000	5000	5000	2000	2000
MIN_COLS_ITER	–	10	10	1	10	1
Running time (total)	41.27	34.38	26.81	27.30	33.75	61.62
Running time (root)	14.28	4.59	3.21	3.27	9.17	43.08
No. of nodes	39	45	43	43	43	41
SPPTWCC calls	158	168	157	157	292	719
No. of routes	5701	5743	5159	5159	4550	4507

Table 5: Testing prematurely stop of SPPTWCC on "medium" problems.

R202 – 25 customers					
LIMIT	–	50000	50000	10000	5000
MIN_COLS_ITER	–	10	1	1	1
Running time (total)	3322.17	256.34	227.46	13.278	7.97
Running time (root)	3316.75	251.44	222.438	9.43	4.34
No. of nodes	5	5	5	5	5
SPPTWCC calls	59	63	63	59	58
No. of routes	6913	6921	6921	6174	5983

Table 6: Testing prematurely stop of SPPTWCC on "tough" problems.

in route quality, but as the quality of the dual variables is low there is practically nothing to loose. And even considering the "easy" problem an improvement is made in the running time indicating that stopping early is an advantage for all problems.

Now we can return to our initial example: R203 with 25 customers. As a test we ran it with LIMIT = 5000 and MIN_COLS_ITER = 1. The result can be seen in figure 5.

```
---------- Solution
Problem R203 with 25 customers is solved
The solution is the following routes:

( 5120) [1041] d - 6 - 5 - 8 - 17 - 16 - 14 - 13 - d

( 5778) [1533] d - 2 - 15 - 23 - 22 - 21 - 4 - 25 - 24 - 3 - 12 - d

( 9416) [1365] d - 18 - 7 - 19 - 11 - 20 - 9 - 10 - 1 - d

---------- Statistics
This program ran on serv3 (hp9000s700).
Total execution time                       147.55 seconds
         (Solving root 8.56 seconds)
Time used in separation                      0.26 seconds
         Cuts generated            2
Accumulated time used in calls of SPPTWCC   14.65 seconds
Time used in largest single SPPTWCC call     0.25 seconds
Branching nodes examined 41 (Veh 1, Arc 20, TW 0)
        (hereof 0 where not feasible)
No of calls to SPPTW 273, Routes generated 19396
Max no of columns selected per SPPTW 200
Prematurely exiting of SPPTWCC enables.
        LIMIT is set to 5000.
        MIN_COLS_ITER is set to 1.
No of multiple customers deleted explicitly 0
IP value 3914
RP value 3816.250
LP value 3798.818
-------------------------------------------------------
```

Figure 5: The result of solving R203 with 25 customers with our algorithm improved the early stopping criteria.

Recall that before the running time was over 13000 seconds, now we are down to around 150 seconds - a reduction by a factor 91. Note how the time spent in the most time consuming SPPTWCC call is reduced from 4447.04 seconds to 0.25 seconds.

To test our algorithm we tried to solve instances from the R2, C2 and RC2 test sets. Their large time windows make even instances with fwe customers difficult to solve. The large time windows result in many feasible routes thereby slowing down the SPPTWCC subsroutine. For the few instances where we know the running time of the original code it is reported in the first column of table 7, where the results are reported. Every instance of R2, C2 and RC2 using the 25 and 50 first customers was run for 30 minutes before execution was stopped. An "**R**" in the second column indicates that execution was stopped while working on the root node of the Branch-and-Bound tree, whereas a boldface integer presents the descendant that the algorithm was working on as the algorithm was stopped.

Especially for large time windows forced early stop results in a drastic decrease of running time. We were able to solve 16 of the demanding R2, C2 and RC2 problems. Comparing with the running times for the algorithm without forced early stop the improvement in performance is huge.

6 Conclusion

Both the column deletion scheme and the forced early stop have proven to be effective ways to cut the execution time of the present state-of-the-art approach for solving VRPTW instances to optimality.

In column deletion we generally observe a significant decrease in running time. It is although essential not to run the columns deletion subroutine too often. Deleting columns too often will force the SPPTWCC subroutine to regenerate a significant amount of routes often. The columns deletion subroutine should be used more dynamically than shown here, but before that is possible further research is necessary.

Instance	25 customers		50 customers	
	Time (before)	Time (now)	Time (before)	Time (now)
R201	14.82	1.92		10.73
R202	3322.17	7.97		272.92
R203	13249.63	147.55	> 10000	**R**
R204		**3**		**R**
R205	325.99	16.69		**93**
R206		**12**		**R**
R207		**3**		**R**
R208	> 10000	**R**		**R**
R209		**3**		**3**
R210		**18**		**R**
R211		**3**		**R**
C201	48.57	3.12	> 10000	208.74
C202	107.93	12.9		**R**
C203	1411.91	33.17		**R**
C204		**R**	> 10000	**R**
C205	28.55	7.66		**R**
C206		21.60		**R**
C207	> 10000	149.53		**R**
C208		80.28		**R**
RC201	8.52	1.29	70.83	47.30
RC202		**35**		**R**
RC203		**14**		**R**
RC204		**R**		**R**
RC205	116.45	72.16		**R**
RC206		**4**		**R**
RC207		**3**		**R**
RC208		**R**		**R**

Table 7: The results of our half-hour test of 25 and 50 customer problems from the test sets R2, C2 and RC2 (using LIMIT = 5000 and MIN_COLS_ITER = 1).

Worth noting on the forced early stop is that it performs very well on large time windows almost without damaging the performance of the algorithm on the small time windows.

Both schemes show that the performance of the column-generation-based VRPTW algorithm can be raised significantly by carefully studing numbers of printouts from running different instances.

References

[CR99] William Cook and Jennifer L. Rich. A parallel cutting-plane algorithm for the vehicle routing problem with time windows. Technical Report TR99-04, Departement of Computational and Applied Mathematics, Rice University, 1999.

[DDS92] Martin Desrochers, Jacques Desrosiers, and Marius Solomon. A new optimization algorithm for the vehicle routing problem with time windows. *Operations Research*, 40(2):342 – 354, March-April 1992.

[GDDS95] Sylvie Gélinas, Martin Desrochers, Jacques Desrosiers, and Marius M. Solomon. A new branching strategy for time constrained routing problems with application to backhauling. *Annals of Operations Research*, 61:91 – 109, 1995.

[Hal92] Karsten Halse. *Modeling and Solving Complex Vehicle Routing Problems.* PhD thesis, Department for Mathematical Modeling, Technical University of Denmark, 1992.

[KDM+97] Niklas Kohl, Jacques Desrosiers, Oli B. G. Madsen, Marius M. Solomon, and François Soumis. k-path cuts for the vehicle routing problem with time windows. Technical Report IMM-REP-1997-12, Department of Mathematical Modelling, Technical University of Denmark, 1997.

[Koh95] Niklas Kohl. *Exact methods for Time Constained Routing and Related Scheduling Problems.* PhD thesis, Department of Mathematical Modelling, Technical University of Denmark, 1995. IMM-PHD-1995-16.

[KRT87] A. W. J. Kolen, A. H. G. Rinnooy Kaan, and H. W. J. M. Trienekens. Vehicle routing with time windows. *Operations Research*, 35(2):266 – 273, March-April 1987.

[Lar99] Jesper Larsen. *Parallellization of the Vehicle Routing Problem with Time Windows*. PhD thesis, Department of Mathematical Modelling, Technical University of Denmark, 1999. IMM-PHD-1999-62.

[LCM00] Jesper Larsen, Jens Clausen, and Oli B. G. Madsen. Using parallel computers to solve the vehicle routing problem with time windows. Technical Report IMM-TP-2000-15, Informatics and Mathematical Modelling, Technical University of Denmark, 2000.