

Primal dual column generation method english language essay

[Linguistics](#), [English](#)



**ASSIGN
BUSTER**

In this chapter we present theoretical and computational developments of the primal-dual column generation method (PDCGM). Firstly, we describe the method and its main components and provide evidence of its convergence. In the second part, we present the results of computational tests aimed to compare the performance of the primal-dual column generation method against the standard column generation (SCG) and the analytic centre cutting plane methods (ACCPM). All the descriptions and results in this chapter closely follow the developments presented in cite{GonGonMun2013}. As discussed briefly in Chapter ef{ch: colgen}, the standard column generation and the analytic centre approaches are extremal strategies, as they are based on optimal solutions but for different objective functions. With the former strategy one obtains an optimal vertex solution at every iteration while with the latter the solution of the barrier problem associated with the localization set in the dual space. In fact, the analytic centre of a feasible set corresponds to the optimal solution of a modified dual problem associated with the RMP. From this point of view, the idea of the primal-dual column generation technique is somewhere in the middle between these two approaches. It relies on solutions that are close-to-optimality (suboptimal), but at the same time not far from the central trajectory in the dual feasible set (well-centred). The contribution of using well-centred suboptimal solutions is twofold. First, fewer inner iterations are usually needed to solve each RMP since we do not require a high accuracy in an initial stage. Hence, the running times per outer iteration is usually reduced at this stage. To understand this point, let us consider Figure ef{fig: strategies} which illustrates the possible dual solutions found by these three

methods and provided to the oracle.

[H]centeringincludegraphics[keepaspectratio= true, clip= true, viewport = 650 700 2700 1800, scale = 0. 14] {figures/strategies}caption{Schematic illustration of solutions provided by the SCG (\circ), the PDCGM (\diamond) and the ACCPM (\square) strategies in the dual space}label{fig: strategies}end{figure}

The area represented by horizontal lines is the localization set and the dashed line represents the best dual bound found so far. The circle denotes the solution obtained by the standard column generation method based on the simplex method which is at the vertex of the feasible region. On the other hand, the square represents the solution obtained by the analytic centre cutting plane method. The central trajectory (in a very particular space) is denoted by the dashed-dotted curved line while the diamond figure illustrates the solution obtained by the primal-dual column generation method. Note that the solution obtained by the PDCGM becomes closer to the optimal facet of the polytope in the dual space as the column generation method approaches the termination. This is explained in the next section. Second, a more stable column generation strategy is likely to be obtained since during the first iterations and when the RMP has not yet gathered enough information, the dual variables are not expected to change dramatically from one iteration to another due to the use of well-centred solutions (close to the central path). The result we expect from this is that a smaller number of outer iterations as well as less total CPU time will be usually required to solve the MP. This strategy known as the primal-dual column generation method was proposed in cite{GonSar96} and is based on suboptimal solutions of the RMPs. A

primal-dual interior point method is used to solve the RMPs, which makes possible obtaining primal-dual feasible solutions which are well-centred in the feasible set, but have a non-zero distance to optimality. Note that this is not achievable if one would like to use a simplex-type method since in general the distance to optimality is not known in advance.

section{Theoretical developments}Following the notation of Chapter ef{ch: colgen}, we consider that a given RMP is represented by

(ef{restrictedmasterproblem})), with optimal primal-dual solution $(\overline{\lambda}, \overline{u})$. Similarly to the standard column generation approach, the primal-dual column generation starts with an initial RMP with enough columns to avoid an unbounded solution. However, at a given outer iteration, instead of solving the problem to optimality, a suboptimal feasible solution $(\tilde{\lambda}, \tilde{u})$ of the current RMP is obtained. This suboptimal solution is defined as follows.

label{def: suboptimal: sol}A primal-dual feasible solution $(\tilde{\lambda}, \tilde{u})$ of the RMP is called suboptimal solution, or ε -optimal solution, if it satisfies

$$0 \leq (c^T \tilde{\lambda} - b^T \tilde{u}) \leq \varepsilon (1 + \|c^T \tilde{\lambda}\|),$$

for some tolerance $\varepsilon > 0$.

We denote by $\tilde{z}_{\text{RMP}} = c^T \tilde{\lambda}$ the objective value corresponding to the suboptimal solution $(\tilde{\lambda}, \tilde{u})$. Since $c^T \tilde{\lambda} \geq c^T \overline{\lambda} = z_{\text{RMP}}$, we have the following inequalities $z_{\text{MP}} \leq z_{\text{RMP}} \leq \tilde{z}_{\text{RMP}}$ and therefore, \tilde{z}_{RMP} is a valid upper bound of the optimal value of the MP. Once the suboptimal solution of the RMP is

obtained, the oracle is called with the dual solution \tilde{u} as a query point. Then, it should return either a value $z_{SP}(\tilde{u}) = 0$, if no columns could be generated using the proposed query point, or a value $z_{SP}(\tilde{u}) < 0$, together with one or more columns to be added to the RMP. Observe that in the second case at least one column can always be generated, as \tilde{u} is dual feasible for the RMP and, hence, all columns already generated must have a non-negative reduced cost. Having defined the concept of suboptimal solutions, we can now show that the bounds provided by using such solutions, are valid in a column generation context. To do so, let us first consider the value $\kappa > 0$ defined as in (eq: kappa: upper). The value of κ depends on the application; however, it is typically a known parameter. According to the following lemma, a lower bound of the optimal value of the MP can still be obtained. It is the classical Lagrangian bound (see e. g. cite{BriLemMeuMicPerVan08, benamor2009}), but derived from a column generation scheme and using a suboptimal solution.

Lemma (subopt: lb) Let $\tilde{z}_{SP} := z_{SP}(\tilde{u})$ be the value of the oracle corresponding to the suboptimal solution $(\tilde{\lambda}, \tilde{u})$. Then, $\kappa \tilde{z}_{SP} + b^T \tilde{u} \leq z^*$.

Proof Let λ^* be an optimal primal solution of the MP. By using (eq: mp: 01) and $\tilde{z}_{SP} \leq 0$, we have that

$$\lambda^* - b^T \tilde{u} = \sum_{j \in N} c_j \lambda_j^* - \sum_{j \in N} \lambda_j^* a_j^T \tilde{u} = \sum_{j \in N} \lambda_j^* (c_j - a_j^T \tilde{u}) \geq \sum_{j \in N} \lambda_j^* \tilde{z}_{SP} \geq \kappa \tilde{z}_{SP}.$$

Therefore, we have that $z^{\star} = c^T \lambda^{\star} \geq \kappa \tilde{z}_{SP} + b^T \tilde{u}$. \square The solution $(\tilde{\lambda}, \tilde{u})$ should also be well-centred in the primal-dual feasible set, in order to provide a more stable dual information to the oracle. We say a point (λ, u) is well-centred if it satisfies

$$\gamma \mu \leq (c_j - u^T a_j) \lambda_j \leq (1/\gamma) \mu, \text{ for all } j \in \overline{N}, \quad \text{label{eq: well-centred: sol}}$$

for some $\gamma \in (0, 1]$, where $\mu = (1/|\overline{N}|) (c^T - u^T A) \lambda$. This resembles our definition of symmetric neighbourhood in [eqref{symmetric: neighbourhood}](#) for particular values of γ , noting that $s_j = c_j - u^T a_j$ and $\lambda_j = x_j$, for every $j \in \overline{N}$. By imposing ([eq: well-centred: sol](#)), we guarantee that the point is not too close to the boundary of the primal-dual feasible set and, hence, the oscillation of the dual solutions will be relatively small. Notice that ([eq: well-centred: sol](#)) is a natural way of stabilizing the dual solutions, if a primal-dual interior point method is used to solve the RMP [cite{Wri97, Gondzio2011}](#). One important observation is that the tolerance ε which controls the distance of $(\tilde{\lambda}, \tilde{u})$ to optimality can be “large” at the beginning of the column generation process, as a very rough approximation of the MP is known at this time. We exploit this fact and during the first iterations, the PDCGM aims to find interesting columns as quickly as possible by solving the RMP to a predefined tolerance. However, solving the RMPs to a loose tolerance is likely to hamper the convergence to the optimal MP so at some point along the process this tolerance needs to be tightened to ensure that

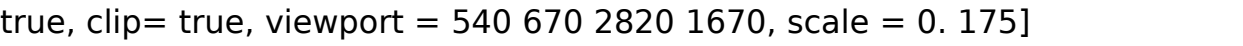
the method converges. The best way we have to identify when the column generation is converging to an optimal solution of the MP is by using the relative gap in the outer iterations, which is given by

$$\text{gap} := \frac{c^T \tilde{\lambda} - (\kappa \tilde{z}_{SP} + b^T \tilde{u})}{1 + |c^T \tilde{\lambda}|},$$

where $\tilde{z}_{SP} := z_{SP}(\tilde{u})$, as defined in Lemma [ef{lemma: subopt: lb}](#). Then, at the end of every outer iteration, we recompute the relative gap, and the tolerance ε used in the PDCGM is updated

$$\varepsilon^k := \min \{ \varepsilon_{\max}, \text{gap}^{k-1} / D \},$$

where $D > 1$ is the [exit{degree of optimality}](#) that relates the tolerance ε^k to the relative gap at iteration $k-1$. In our theoretical developments and computational experiments, we consider D as a fixed parameter. Also, a threshold ε_{\max} is used so that the suboptimal solution is not too far away from the optimum. Note that the update of the tolerance after reaching the break point is done gradually and we do not require the method to solve every RMP to optimality apart from the last iterations. By using this dynamically adjusted tolerance, we expect to reduce the problems arising when using the standard column generation, namely the heading-in and the tailing-off effects.



[figures/deg_opt](#)

[caption{Results with the PDCGM for the VRPTW - Role of the dynamically adjusted tolerance}](#)

[label{fig: deg_opt}](#)

In [Figure ef{fig: deg_opt}](#), we demonstrate the behaviour of the PDCGM when solving a random instance of the vehicle routing problem with time windows. The

figure illustrates the changes in the relative duality gap over the iterations. Note that a relative gap greater than 1 occurs when $\kappa \tilde{z}_{SP} + b^T \tilde{u} < -1$. The horizontal dashed line denotes the break point when the method switches from using a loose tolerance ε_{\max} to a tighter tolerance, namely gap^{k-1}/D . During all the process, we pass to subproblems the well-centred solutions provided by the use of the primal-dual interior point method and considering the symmetric neighbourhood. It is important to emphasize that unlike the standard approach, $\tilde{z}_{SP} = 0$ does not suffice to terminate the column generation process. Indeed $\tilde{\lambda}$ is feasible for the MP while \tilde{u} is not and therefore there may still be a difference between $c^T \tilde{\lambda}$ and $b^T \tilde{u}$. Lemma ef{lemma: zsp: 0} below shows that the gap is still reduced in this case, and the progress of the algorithm is guaranteed. It is important to emphasize that unlike the standard approach, $\tilde{z}_{SP} = 0$ does not suffice to terminate the column generation process since there may still be a difference between $c^T \tilde{\lambda}$ and $b^T \tilde{u}$. Lemma ef{lemma: zsp: 0} below shows that the gap is still reduced in this case, and the progress of the algorithm is guaranteed.

Let $(\tilde{\lambda}, \tilde{u})$ be the suboptimal solution of the RMP, found at iteration k with tolerance $\varepsilon^k > 0$. If $\tilde{z}_{SP} = 0$, then the new relative gap is strictly smaller than the previous one, i.e., $\text{gap}^k < \text{gap}^{k-1}$.

We have that $\tilde{z}_{RMP} = c^T \tilde{\lambda}$ is an upper bound of the optimal solution of the MP. Also, from Lemma ef{lemma: subopt: lb} we

obtain the lower bound $b^T \tilde{u}$, since $\tilde{z}_{SP} = 0$. Hence, the gap in the current iteration is given by $gap^k = \frac{c^T \tilde{\lambda} - b^T \tilde{u}}{1 + \|c^T \tilde{\lambda}\|}$. Notice that from Definition ef{def: suboptimal: sol}, the right-hand side of this equality is less or equal than ε^k , the tolerance used to obtain $(\tilde{\lambda}, \tilde{u})$. Hence, $gap^k \leq \varepsilon^k$. We have two possible values for ε^k . If $\varepsilon^k = \varepsilon_{\max}$, then by (ef{eq: suboptimal: tolerance: gap}) $gap^{k-1} \geq D \varepsilon^k > \varepsilon^k$. Otherwise, $\varepsilon^k = gap^{k-1} / D$ and, again, $gap^{k-1} > \varepsilon^k$ which completes the proof. end{proof} Algorithm ef{alg: pdcgm} summarizes the above discussion. Notice that the primal-dual column generation method has a simple algorithmic description, similar to the standard approach (compare with Algorithm ef{alg: col_gen}). Thus, it can be implemented in the same level of difficulty if a primal-dual interior point solver is readily available. Notice that κ is known in advance and problem dependent. Also, the upper bound of the RMP, \tilde{z}_{RMP} , may slightly increase from one iteration to another due to the use of suboptimal solutions and, hence, we store the best value found so far in UB (Step ef{ub: store} in Algorithm ef{alg: pdcgm}).

egin{algorithm}caption{The Primal-Dual Column Generation Method}label{alg: pdcgm}extbf{Set:} $\mathit{LB} = -\infty$, $\mathit{UB} = \infty$, $gap = \infty$, $\varepsilon = 0.5$;egin{algorithmic}

[1]WHILE{ $(gap \geq \delta)$ }STATE{find a well-centred ε -optimal solution $(\tilde{\lambda}, \tilde{u})$ of the RMP;}STATE{ $\mathit{UB} = \min\{\mathit{UB}, \tilde{z}_{RMP}\}$;} label{ub: store}STATE{call the oracle

with the query point \tilde{u} and set $\tilde{z}_{SP} := z_{SP}(\tilde{u})$

$$\text{STATE}\{\text{mbox{LB}} = \max\{\text{mbox{LB}}, \kappa \tilde{z}_{SP} + b^T \tilde{u}\}\};$$

$$\text{STATE}\{\text{gap} = (\text{mbox{UB}} - \text{mbox{LB}}) / (1 + |\text{mbox{UB}}|)\};$$

$$\text{STATE}\{\text{varepsilon} = \min\{\text{varepsilon}_{\max}, \text{gap} / D\}\};$$

$$\text{STATE}\{\text{if } (\tilde{z}_{SP} < 0) \text{ then add the new columns to the}$$

RMP;\}ENDWHILEend{algorithmic}end{algorithm}

Since the PDCGM relies on suboptimal solutions of each RMP, it is important to provide guarantees that it is a valid column generation procedure, *i. e.*, a finite iterative process that delivers an optimal solution of the MP. Even though the optimality tolerance varepsilon decreases geometrically in the algorithm, there is a special case in which the subproblem value is zero, which would cause the method to stall. Fortunately, by using Lemma [ef{lemma: zsp: 0}](#) we can guarantee the method still converges to the optimal solution of the MP. The proof of convergence is given in Theorem [ef{theorem: finite}](#).

[ef{theorem: finite}](#) [label{theorem: finite}](#) Let z^{\star} be the optimal value of the MP. Given $\delta > 0$, the primal-dual column generation method converges in a finite number of steps to a primal feasible solution $\hat{\lambda}$ of the MP with objective value \tilde{z} that

$$|\tilde{z} - z^{\star}| < \delta (1 + |\tilde{z}|). \quad \text{label{eq: theorem: gap}}$$

[ef{theorem: gap}](#)

[egin{proof}](#) Consider an arbitrary iteration k of the primal-dual column generation method, with corresponding suboptimal solution $(\tilde{\lambda}, \tilde{u})$. After calling the oracle, two situations may occur:

- $\tilde{z}_{SP} < 0$ and new columns have been generated. These columns correspond to dual constraints of the MP that are violated by the dual point \tilde{u} .

Since the columns are added to the RMP, the corresponding dual constraints will not be violated in the next iterations. Therefore, it guarantees the progress of the algorithm. Also, this case can only happen a finite number of times, as there are a finite number of columns in the MP. item $\tilde{z}_{SP} = 0$ and no columns have been generated. If additionally we have $\varepsilon^k < \delta$, then from Lemma ef{lemma: zsp: 0} the gap in the current iteration satisfies $\text{gap}^k < \delta$, and the algorithm terminates with the suboptimal solution $(\tilde{\lambda}, \tilde{u})$.

Otherwise, we also know from Lemma ef{lemma: zsp: 0} that the gap is still reduced, and although the RMP in the next iteration will be the same, it will be solved to a tolerance $\varepsilon^{k+1} < \varepsilon^k$. Moreover, the gap is reduced by a factor of $1/D$ which is less than 1 and, hence, after a finite number of iterations we obtain a gap less than δ .

end{enumerate}At the end of the iteration, if the current gap satisfies

$\text{gap}^k < \delta$, then the algorithm terminates and we have

$$\frac{|\tilde{z}_{RMP} - (\kappa \tilde{z}_{SP} + b^T \tilde{u})|}{1 + |\tilde{z}_{RMP}|} < \delta.$$

Since $\kappa \tilde{z}_{SP} + b^T \tilde{u} \leq z^*$, the inequality (ef{eq: theorem: gap}) is satisfied with $\tilde{z} = \tilde{z}_{RMP}$.

The primal solution $\tilde{\lambda}$ leads to a primal

feasible solution of the MP, given by $\hat{\lambda}_j = \tilde{\lambda}_j$,

for all $j \in \overline{N}$, and $\hat{\lambda}_j = 0$, otherwise. If gap^k

$\geq \delta$, a new iteration is carried out and we have one of the above

situations again. end{proof}Having presented a proof of convergence for the

PDCGM, it is important to give some remarks about its implementation. As

requested by (ef{eq: well-centred: sol}), the suboptimal solutions are well-

centred points in the primal-dual feasible set. This contributes to the stabilization of the dual points and, hence, reduces the number of outer iterations in general. In our implementation, each RMP is solved by the interior point solver HOPDM cite{gondzio1995}. It keeps the iterates inside a neighbourhood of the central path, which has the form (ef{eq: well-centred: sol}). To achieve this, the solver makes use of multiple centrality correctors cite{gondzio1996multiple, colombo2008}. An efficient warmstarting technique is essential for a good performance of a column generation technique based on an interior point method, such as the PDCGM. Throughout the column generation process, closely-related problems are solved, as the RMP in a given iteration differs from the RMP of the previous iteration by merely a few columns. Hence, this similarity should be exploited in order to reduce the computational effort of solving a sequence of problems. In our implementation of PDCGM, we rely on the warmstarting technique available in the solver HOPDM cite{Gon98}. The main idea of this method consists of storing a close-to-optimality and well-centred iterate when solving a given RMP. After a modification is carried out on the RMP, the stored point is adjusted to create a full-dimensional initial point to start from. Warmstarting is a key aspect of a successful column generation implementation and therefore, in Chapter ef{ch: warmstarting} we include a complete analysis of this feature and describe a new warmstarting strategy in this context. Notice that a primal-dual interior point method is well-suited for the implementation of the PDCGM. In fact, (standard) simplex-based methods cannot straightforwardly provide suboptimal solutions which are well-centred in the dual space. Instead, the primal and dual solutions are always on the

boundaries of their corresponding feasible sets. Besides, there is no control on the infeasibilities of the solutions before optimality is reached in a simplex method. Before moving to the next section of the chapter, it is important to clarify that the PDCGM is more than a strategy which replaces the simplex method with a primal-dual interior point method in column generation. It has to be understood as a new column generation strategy that exploits the rich information provided by a primal-dual interior point method (distance to optimality) to obtain suboptimal solutions as needed. Also, the dynamic adjustment of the tolerance ensures that the method does not stall and converges to the optimal solution. Finally, the method requires only two parameters to be set which have straightforward meanings:

ϵ_{\max} defines the tolerance for the initial iterations when a loose approximation of the MP is at hand, while D , determines how fast we would like to approach the optimality.

section{Computational study}
label{sec: pdcgm: results}

In this section we present extensive numerical results obtained by using different column generation strategies for different applications. Note that the results included here follow a different presentation than the ones discussed in cite{GonGonMun2013}. As benchmarks we have chosen three different applications which are well known in the column generation literature and have been described already in Chapter ef{ch: formulations}: the cutting stock problem (CSP), the vehicle routing problem with time windows (VRPTW), and the capacitated lot sizing problem with setup times (CLSPST). For each application, we have implemented three different column generation strategies which were already described. A brief summary of each strategy and some

implementation considerations are given below:

- Standard column generation (SCG): each RMP is solved to optimality by a simplex-type method to obtain an extreme optimal dual solution. The implementation closely follows the steps presented in Algorithm ef{alg: col_gen}. We have used one of the best available linear programming solvers, CPLEX cite{CPLEX} to obtain such a solution. Preliminary tests using the default settings for each solver show that the primal simplex method is slightly better than the dual method as the optimal basis remains primal feasible from one outer iteration to another. The overall performance using the barrier method (with crossover) was inferior to the other two methods.
- Primal-dual column generation (PDCGM): the suboptimal solutions of each RMP are obtained by using the interior point solver HOPDM cite{gondzio1995}, which is able to efficiently provide well-centred suboptimal dual points. The algorithm has been already described in Algorithm ef{alg: pdcgm}.
- Analytic centre cutting plane (ACCPM): the dual point at each iteration is an approximate analytic centre of the localization set associated with the current RMP described in Section ef{sec: cg: strategies}. The applications were implemented on top of the open-source solver OBOE/COIN cite{OBOE}, a state-of-the-art implementation of the analytic centre strategy with additional stabilization terms cite{babonneau2007}.

For each application and for the aforementioned column generation strategies, the subproblems are solved using the same source-code. We provide more details of the oracle solvers used in each application later. Also, the SCG and the PDCGM are initialized with the same columns and, hence, have the same initial RMP. The ACCPM

requires an initial dual point to start from, instead of a set of initial columns. After preliminary tests, we have chosen initial dual points that led to a better performance of the method on average. We have used different initial dual points for each application, as specified later. All the computational experiments in this section have been obtained using a computer with processor Intel Core i2 Duo 2.26 Ghz, 4 GB RAM, and Linux operating system. For each of the strategies, we stop the column generation procedure when the relative gap becomes smaller than the default accuracy $\delta = 10^{-6}$. The purpose of comparing the PDCGM against the SCG is to give an idea of how much can be gained in overall performance in relation to the standard approach using extreme dual solutions without any stabilization. It is important to note that due to degeneracy issues an extreme optimal dual solution obtained by the SCG can be at any vertex of the optimal facet of the feasible polyhedra. Undoubtedly, it would be interesting to consider stabilized versions of the standard column generation in our computational comparisons. However, the lack of publicly available codes of stabilized versions discouraged us to include them. For the interested reader, available comparisons between standard and stabilized column generation are available in the literature for the same applications cite{Rousseau2007, BriLemMeuMicPerVan08, benamor2009}. Additionally, we have included the ACCPM in our experiments for being a strategy that also relies on an interior point method (although essentially different) providing well-centred dual solutions. Before continuing, the performance of interior point methods in a column generation by solving each RMP to optimality by a state-of-the-art solver (HOPDM) has been tested. The results obtained were not better than

the ones obtained by the PDCGM which shows that an appropriate use of an interior point method is essential for its success in the column generation context. In the applications addressed in this thesis, the column generation schemes are obtained by applying Dantzig-Wolfe decomposition (DWD) to the corresponding integer programming formulations cite{dantzig1960, vanderbeck2000}. In each application, the decomposition leads to an integer MP and also an integer (pricing) subproblem. As shown in Chapter ef{ch: formulations}, we relax the integrality of the variables in the integer MP and then solve it by column generation, which gives a lower bound of the optimal value of the original formulation. To obtain an integer solution, it would be necessary to combine the column generation with a branch-and-bound search, which is called a branch-and-price method cite{barnhart1998, LubDes05}. However, this combination is out of the scope of this thesis, as we are concerned with the behaviour of the column generation strategies. A very recent attempt of combining the primal-dual column generation method in a branch-and-price context to solve the vehicle routing problem with time windows can be found in cite{MunGon2012}. This study shows encouraging results when compared to the state-of-the-art method to solve this class of problems. subsection{Cutting stock problem}To analyse the performance of the different column generation strategies addressed here, we have initially selected \$262\$ instances from the one-dimensional CSP literature. All the instances were obtained from url{http://www. math. tu-dresden. de/~capad/}. For this application, the initial RMP consists of columns generated by \$m\$ homogeneous cutting patterns, which corresponds to selecting only one piece per pattern as many times as possible without

violating the width W . In the ACCPM approach and after testing with different values, we have used the initial guess $u^0 = 0.5$ which has provided the best results for this strategy. The knapsack problem is solved using a branch-and-bound method described in cite{leao2009}, the implementation of which was provided by the author. paragraph{Adding one column to the RMP per iteration}In the first set of numerical experiments we consider that only one column is generated by the subproblem solver at each iteration. We have classified the instances into different classes. Table ef{csp-table-average} presents for each class and strategy: the average number of outer iterations (ite), the average CPU time spent in the oracle in seconds (or(s)) and the average CPU time required for the column generation procedure in seconds (tot(s)). The number of instances per class is shown in column $\#$. The last row (ALL) presents the average results considering the 262 instances. Additionally, the last four columns show the ratio between the extreme strategies and the PDCGM. From the results of our first set of experiments, it seems clear that the PDCGM does not offer any savings in terms of CPU time compared to the SCG when ``easy" instances are solved (classes MTP0xJES, MTP0, hard28, 7hard, 53NIRUPs and gau3). Note that these classes are considered ``easy" since the total time required to solve the subproblems (oracle time) for all the strategies is less than 1 second in average. Among these classes, the SCG is the strategy that offers the best performance if total CPU time is considered. Note that the value of m (number of different widths) included in this class varies from 20 to 189. For all these classes, the PDCGM is the strategy that achieves the smallest number of iterations on average. Similar results are obtained when

considering the class with more instances (mX), where m varies from 197 to 200. For this class, the oracle solver requires more time to find the columns and therefore, the savings in terms of iterations start to pay off. Finally, if we consider class U, with m varying from 15 to 585, the best overall performance is provided by the PDCGM in both performance measures, number of iteration and total CPU time. Note that for this class the three strategies require a considerable time to solve the subproblems and therefore, savings in number of iterations have an important impact on the total CPU time. The ACCPM does not offer any benefit in any class when compared to the other two strategies. If all the instances are considered, the PDCGM offers the best overall performance. Observe that the RMPs solved at each outer iteration are actually small/medium size linear programming problems. The number of columns in the last RMP is approximately the number of initial columns plus the number of outer iterations. Note that for the SCG the time spent in solving the RMPs is very small in relation to the time required to solve the subproblems, regardless the size of the instances. It happens because the simplex method implementation available in CPLEX is very efficient on solving/reoptimizing these linear programming problems. For the PDCGM and the ACCPM, the proportion of the total CPU time required to solve the RMP and the oracle varies according to the size of the instances.

Average results on 262 instances of the CSP for the SCG, PDCGM and ACCPM adding one column at a time.

| | SCG | PDCGM | ACCPM |
|--------------|------|-------|-------|
| Iterations | 1000 | 1000 | 1000 |
| CPU time (s) | 1000 | 1000 | 1000 |

6&632. 1&636. 8&782. 7&134. 4&154. 7&871. 5&582. 9&694. 7&1. 2&4.
1&1. 1&4. 5mX&145&803. 6&4. 1&5. 2&504. 2&4. 2&9. 2&651. 6&6. 3&20.
5&1. 6&0. 6&1. 3&2. 2MTP0xjES&3&379. 7&0. 8&0. 9&244. 7&0. 8&1.
6&294. 0&0. 8&2. 2&1. 6&0. 6&1. 2&1. 4MTP0&5&383. 0&0. 7&0. 8&264.
0&0. 6&1. 5&303. 6&0. 7&2. 4&1. 5&0. 5&1. 2&1. 6hard28&28&535. 7&0.
3&0. 8&386. 4&0. 5&2. 9&475. 8&0. 7&6. 1&1. 4&0. 3&1. 2&2.

7ottomruleend{tabular}%label{csp-table-
 average}end{adjustwidth}end{table}%paragraph{Adding k -best columns
 to the RMP per iteration}The knapsack solver used to solve the CSP
 subproblems is able to obtain not only the optimal solution, but also the k -
 best solutions for a given $k > 0$. Hence, we can generate up to k
 columns in one call to the oracle to be added to the RMP. It usually improves
 the performance of a column generation procedure, since more information
 is gathered at each iteration. With this in mind, we carry out a second set of
 experiments for the CSP in which we have tested these strategies for three

different values of k .

Average results on 262 instances of the CSP for the SCG, PDCGM and ACCPM adding up to k columns at a time.

| | SCG | PDCGM | ACCPM | SCG/PDCGM | ACCPM/PDCGM |
|----------|-------|-------|-------|-----------|-------------|
| U | 304.4 | 311.9 | 315.0 | 176.2 | 57.4 |
| mX | 219.6 | 1.9 | 2.5 | 143.0 | 1.7 |
| MTP0xJES | 82.0 | 0.3 | 0.3 | 60.0 | 0.3 |
| MTP0 | 84.2 | 0.2 | 0.3 | 61.8 | 0.3 |
| hard28 | 149.1 | 0.2 | 0.5 | 110.7 | 0.3 |
| 7hard | 85.3 | 0.1 | 0.2 | 70.0 | 0.1 |
| 53NIRUPs | 82.6 | 0.1 | 0.2 | 59.2 | 0.1 |
| gau3 | 25.0 | 0.0 | 0.0 | 21.0 | 0.0 |
| ALL | 182.3 | 4.3 | 4.3 | 182.3 | 4.3 |

3} & extbf{24. 9} & extbf{25. 5} & extbf{120. 2} & extbf{5. 4} & extbf{7.
 3} & extbf{290. 0} & extbf{28. 2} & extbf{65. 4} & extbf{1. 5} & extbf{3.
 5} & extbf{2. 4} & extbf{9. 0} midrulemulticolumn{1}{c}{multirow{9}{*}
 {extbf{50}}}} & U & 186. 8 & 223. 8 & 228. 2 & 103. 2 & 64. 0 & 69. 7 &
 277. 6 & 106. 4 & 372. 5 & 1. 8 & 3. 3 & 2. 7 & 5. 3 multicolumn{1}{c}{} &
 mX & 109. 8 & 3. 9 & 4. 5 & 89. 1 & 4. 2 & 7. 6 & 408. 4 & 23. 4 & 223. 2 &
 1. 2 & 0. 6 & 4. 6 & 29. 2 multicolumn{1}{c}{} & MTP0xJES & 35. 3 & 0. 3 &
 0. 4 & 35. 0 & 0. 6 & 0. 8 & 122. 3 & 1. 9 & 4. 8 & 1. 0 & 0. 5 & 3. 5 & 5. 7
 multicolumn{1}{c}{} & MTP0 & 31. 8 & 0. 3 & 0. 3 & 34. 6 & 0. 5 & 0. 8 &
 137. 8 & 1. 7 & 7. 1 & 0. 9 & 0. 4 & 4. 0 & 9. 2 multicolumn{1}{c}{} &
 hard28 & 66. 5 & 0. 6 & 0. 9 & 68. 8 & 0. 9 & 2. 4 & 279. 0 & 4. 4 & 57. 3 &
 1. 0 & 0. 4 & 4. 1 & 23. 9 multicolumn{1}{c}{} & 7hard & 37. 0 & 0. 2 & 0.
 3 & 42. 4 & 0. 4 & 0. 9 & 183. 1 & 1. 6 & 14. 6 & 0. 9 & 0. 4 & 4. 3 & 16. 6
 multicolumn{1}{c}{} & 53NIRUPs & 33. 8 & 0. 2 & 0. 3 & 35. 8 & 0. 3 & 0. 7
 & 148. 8 & 1. 3 & 12. 6 & 0. 9 & 0. 4 & 4. 2 & 17. 4 multicolumn{1}{c}{} &
 gau3 & 13. 0 & 0. 0 & 0. 0 & 20. 0 & 0. 0 & 0. 1 & 84. 0 & 0. 2 & 0. 7 & 0. 7 &
 0. 2 & 4. 2 & 5. 7 midrulemulticolumn{1}{c}{} & extbf{ALL} & extbf{91. 0}
 & extbf{19. 4} & extbf{20. 1} & extbf{74. 1} & extbf{7. 4} & extbf{10. 0}
 & extbf{316. 4} & extbf{21. 9} & extbf{161. 2} & extbf{1. 2} & extbf{2. 0}
 & extbf{4. 3} & extbf{16. 1} midrulemulticolumn{1}{c}{multirow{9}{*}
 {extbf{100}}}} & U & 137. 6 & 248. 5 & 252. 5 & 85. 2 & 72. 1 & 79. 3 &
 291. 6 & 158. 1 & 548. 3 & 1. 6 & 3. 2 & 3. 4 & 6. 9 multicolumn{1}{c}{} &
 mX & 84. 8 & 8. 4 & 9. 0 & 76. 4 & 9. 6 & 14. 9 & 464. 2 & 68. 1 & 468. 8 &
 1. 1 & 0. 6 & 6. 1 & 31. 5 multicolumn{1}{c}{} & MTP0xJES & 22. 7 & 0. 4 &
 0. 5 & 28. 7 & 1. 0 & 1. 3 & 131. 3 & 3. 6 & 8. 3 & 0. 8 & 0. 4 & 4. 6 & 6. 6

multicolumn{1}{c}{} & MTP0 & 25.4 & 0.5 & 0.5 & 28.4 & 0.8 & 1.1 & 144.0 & 3.7 & 11.0 & 0.9 & 0.5 & 5.1 & 9.7 multicolumn{1}{c}{} & hard28 & 48.2 & 1.3 & 1.5 & 57.1 & 2.2 & 4.2 & 304.1 & 13.3 & 105.5 & 0.8 & 0.4 & 5.3 & 25.2 multicolumn{1}{c}{} & 7hard & 27.0 & 0.5 & 0.6 & 37.4 & 0.9 & 1.5 & 198.3 & 4.3 & 35.7 & 0.7 & 0.4 & 5.3 & 23.6 multicolumn{1}{c}{} & 53NIRUPs & 23.5 & 0.4 & 0.5 & 30.5 & 0.7 & 1.2 & 158.4 & 3.4 & 25.7 & 0.8 & 0.4 & 5.2 & 21.4 multicolumn{1}{c}{} & gau3 & 8.0 & 0.0 & 0.0 & 17.0 & 0.1 & 0.2 & 87.0 & 0.3 & 3.6 & 0.5 & 0.2 & 5.1 & 21.1 midrule multicolumn{1}{c}{} & extbf{ALL} & extbf{68.8} & extbf{23.9} & extbf{24.6} & extbf{63.0} & extbf{11.3} & extbf{15.1} & extbf{353.6} & extbf{52.1} & extbf{319.1} & extbf{1.1} & extbf{1.6} & extbf{5.6} & extbf{21.2} ottomrule end{tabular}%label{csp-table-k_columns}end{adjustwidth}end{table}%In Table ef{csp-table-k_columns},

we present the results obtained by adding more than one column at each iteration. For the ``easy'' classes, the SCG is more efficient than the PDCGM and the ACCPM, regardless the number of columns added at each iteration. Similar results are obtained when class mX is considered. However, when instances in class U are solved, the PDCGM is on average more efficient than the SCG and the ACCPM in terms of both outer iterations and CPU time. For instance, if we consider $k = 100$, the PDCGM is 3.2 times faster than the SCG and 6.9 times faster than the ACCPM. Similar results are observed when all instances are considered. Again for $k = 100$, the PDCGM is 1.6 times faster than the SCG and 21.1 times faster than the ACCPM on average. The results indicate that the best overall strategy to solve this selection of 262 instances is the PDCGM with $k = 10$, which is on

average \$2.8\$ and \$9.0\$ times faster than the best result found with the SCG (\$k=50\$) and the ACCPM (\$k=10\$), respectively. Clearly, the behaviour of the ACCPM is adversely affected by the number of columns added at a time, as the number of iterations and the CPU time required for solving the RMPs are considerably increased for larger values of \$k\$. The main reason for this behaviour is that the localization set may be drastically changed from one outer iteration to another if many columns are added. Hence, finding the new analytic centre can be very expensive in this case. A discussion about the warmstarting strategy proposed for the ACCPM is included in Chapter ef{ch: warmstarting}. To conclude this initial set of experiments, it is fair to say that the PDCGM outperforms the other two strategies because it is considerably more efficient in solving the difficult instances in class U. In order to study whether this relative performance can be extended to even larger instances, we have performed a second set of experiments. Additionally, and in order to study the impact of the size of the problems on the different strategy behaviours we have further selected \$14\$ large instances from [url{http://www. math. tu-dresden. de/~capad/}](http://www.math.tu-dresden.de/~capad/) and compared the performance of the three column generation approaches. These instances have \$m\$ varying from \$615\$ to \$1005\$, which leads to larger restricted master problems and also larger subproblems. Table ef{csp-table-very_large} shows the results of this experiment when \$100\$ columns are added per iteration. In all cases, the PDCGM is faster and requires fewer iterations than the SCG and the ACCPM, which supports the conclusion that the relative performance of the PDCGM is improved as the instances become larger and more difficult.

Results on 14 large instances of the CSP for the SCG, PDCGM and ACCPM adding up to 100 columns at a time.

| | SCG | PDCGM | ACCPM | SCG/PDCGM | ACCPM/PDCGM | m | ite | or(s) | tot(s) | ite | or(s) | tot(s) | ite | or(s) | tot(s) |
|------|------|-------|-------|-----------|-------------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| 1005 | 548 | 12760 | 12947 | 293 | 5545 | 5678 | 762 | 10054 | 21254 | 1.9 | 2.3 | 2.6 | 3.7 | U09513 | 975 |
| 518 | 9741 | 9904 | 267 | 4169 | 4277 | 779 | 7404 | 19362 | 1.9 | 2.3 | 2.9 | 4.5 | U09528 | 945 | 541 |
| 9011 | 9173 | 276 | 4811 | 4924 | 740 | 6586 | 15920 | 2.0 | 1.9 | 2.7 | 3.2 | U09543 | 915 | 506 | 7676 |
| 7798 | 263 | 3624 | 3724 | 723 | 5255 | 13449 | 1.9 | 2.1 | 2.7 | 3.6 | U09558 | 885 | 482 | 5479 | 5585 |
| 265 | 2631 | 2730 | 683 | 4222 | 10861 | 1.8 | 2.0 | 2.6 | 4.0 | U09573 | 855 | 473 | 4694 | 4771 | 230 |
| 1980 | 2054 | 672 | 3732 | 9794 | 2.1 | 2.3 | 2.9 | 4.8 | U09588 | 825 | 467 | 4876 | 4950 | 247 | 1574 |
| 1649 | 658 | 3983 | 9376 | 1.9 | 3.0 | 2.7 | 5.7 | U09603 | 795 | 465 | 3894 | 3962 | 237 | 1598 | 1668 |
| 627 | 3055 | 7504 | 2.0 | 2.4 | 2.6 | 4.5 | U09618 | 765 | 424 | 2773 | 2830 | 203 | 1042 | 1092 | 617 |
| 2156 | 6467 | 2.1 | 2.6 | 3.0 | 5.9 | U09633 | 735 | 432 | 2833 | 2878 | 217 | 912 | 969 | 595 | 1751 |
| 5308 | 2.0 | | | | | | | | | | | | | | |

3. 0 & 2. 7 & 5. 5 U09648 & 705 & 424 & 2611 & 2659 & 209 & 808 & 857 &
 582 & 1403 & 4466 & 2. 0 & 3. 1 & 2. 8 & 5. 2 U09663 & 675 & 381 & 2156
 & 2187 & 202 & 613 & 654 & 534 & 1074 & 3325 & 1. 9 & 3. 3 & 2. 6 & 5. 1
 U09678 & 645 & 376 & 1745 & 1775 & 173 & 387 & 418 & 542 & 1043 &
 3395 & 2. 2 & 4. 3 & 3. 1 & 8. 1 U09693 & 615 & 384 & 1324 & 1347 & 165
 & 401 & 427 & 520 & 876 & 2773 & 2. 3 & 3. 2 & 3. 2 & 6. 5

ottomruleend{tabular}%label{csp-table-

very_large}end{adjustwidth}end{table}%subsection{Vehicle routing

problem with time windows}label{sub: pdcgm: vrptw}In order to test the
 behaviour of the different column generation strategies in the VRPTW, we
 have selected \$87\$ instances from the literature ([url{http://www2.imm.dtu.dk/~jla/solomon.html}](http://www2.imm.dtu.dk/~jla/solomon.html)), which were originally proposed in

[cite{solomon1987}](#). The initial columns of the RMP have been generated by
 \$n\$ single-customer routes which correspond to assigning one vehicle per
 customer. In the ACCPM approach, we have considered the initial guess
 $u^0 = 100.0$ e\$ which after testing various settings has proven to be
 the choice which gives the best overall results for this application. Note that
 although several algorithms are available in the literature for solving the
 pricing problem (see [cite{irnich2005}](#) for a survey), solving it to optimality
 may require a relatively large CPU time, especially when the time windows
 are wide. As a consequence, a relaxed version is solved in practice, in which
 non-elementary paths are allowed (i. e., paths that visit the same
 customer more than once). Even though the lower bound provided by the
 column generation scheme may be slightly worse in this case, the CPU time
 to solve the subproblem is considerably reduced. In all our implementations,

the subproblem is solved by our own implementation of the bounded bidirectional dynamic programming algorithm proposed in cite{righini2009}, with state-space relaxation and identification of unreachable nodes cite{feillet2004}. For more details about this implementation, we refer the reader to cite{MunGon2012}. paragraph{Adding one column to the RMP}In Table ef{vrptw-table-average} we compare the performance of the three strategies when only one column is added to the RMP at each iteration. For each class and strategy we present: the number of outer iterations (ite), the average CPU time to solve the subproblems in seconds (or(s)) and the average total CPU time required for the column generation in seconds (tot(s)). Column \$\$\$ contains the number of instances per class. The last row (ALL) shows the average results considering the \$87\$ instances. In the last four columns, the ratios between the extreme strategies and the PDCGM in terms of outer iterations and total CPU time, are presented. The instances are grouped in terms of the distribution of the customers (C: cluster; R: random; RC: a combination of both) and number of customers (\$25\$, \$50\$ and \$100\$). For instance, class C50 contains instances in which \$50\$ customers are distributed in clusters. For all the classes, the PDCGM shows the best average performance in the number of iterations and total CPU time compared with the other two strategies. When the size of the instances increases, the difference between the SCG and the other two strategies increases as well, with the SCG being the one which shows the worst overall performance. Considering the \$87\$ instances, the PDCGM is on average \$3.6\$ and \$1.3\$ times faster than the SCG and the ACCPM, respectively. Notice that, differently from what was observed on the CSP results, the CPU time

| | | | | | | | | | | | | |
|-------------------|---------------|----------------|---------------------|--------------------|---------------|--------------------------|---------------------|----------------|---------|--------|---------|--|
| 1 | in} | centering | egin{tabular} | {cccccccccccccccc} | opr | ule& & multicolumn{3}{c} | | | | | | |
| { | extbf{SCG}} | } | & multicolumn{3}{c} | { | extbf{PDCGM}} | } | & multicolumn{3}{c} | | | | | |
| { | extbf{ACCPM}} | } | & multicolumn{2}{c} | { | in | extbf{SCG/PDCGM}} | } | & | | | | |
| multicolumn{2}{c} | { | in | extbf{ACCPM/PDCGM}} | } | midrule | extbf{class} | } | & # | & | | | |
| extbf{ite} | } | & extbf{or(s)} | } | & extbf{tot(s)} | } | & extbf{ite} | } | & extbf{or(s)} | } | & | | |
| extbf{tot(s)} | } | & extbf{ite} | } | & extbf{or(s)} | } | & extbf{tot(s)} | } | & extbf{ite} | } | & | | |
| extbf{tot(s)} | } | & extbf{ite} | } | & extbf{tot(s)} | } | midrule | C25 | & 9 | & 142.3 | & 1.0 | & | |
| 1.0 | & 35.7 | & 0.3 | & 0.3 | & 53.6 | & 0.3 | & 0.4 | & 4.0 | & 3.1 | & 1.5 | & 1.1 | R25 | |
| & 12 | & 77.8 | & 0.3 | & 0.4 | & 52.9 | & 0.2 | & 0.3 | & 146.0 | & 0.2 | & 0.4 | & 1.5 | | |
| & 1.1 | & 2.8 | & 1.1 | RC25 | & 8 | & 85.3 | & 1.1 | & 1.2 | & 57.1 | & 0.8 | & 0.9 | & | |
| 107.0 | & 1.0 | & 1.2 | & 1.5 | & 1.3 | & 1.9 | & 1.4 | C50 | & 9 | & 446.7 | & 32.1 | & | |
| 32.3 | & 60.3 | & 4.4 | & 4.5 | & 74.6 | & 4.4 | & 4.6 | & 7.4 | & 7.1 | & 1.2 | & 1.0 | | |
| R50 | & 12 | & 211.6 | & 12.2 | & 12.3 | & 122.7 | & 4.8 | & 5.1 | & 214.9 | & 6.4 | & 6.8 | & 1.7 | |
| & 2.4 | & 1.8 | & 1.3 | RC50 | & 8 | & 193.8 | & 18.2 | & 18.3 | & 115.5 | & 9.8 | & 10.1 | & 182.6 | |
| & 12.7 | & 13.1 | & 1.7 | & 1.8 | & 1.6 | & 1.3 | C100 | & 9 | & | | | | |
| 1049.7 | & 333.8 | & 334.8 | & 115.8 | & 51.9 | & 52.3 | & 127.9 | & 50.0 | & 50.4 | | | | |
| & 9.1 | & 6.4 | & 1.1 | & 1.0 | R100 | & 12 | & 700.6 | & 549.0 | & 549.7 | & 260.2 | & | | |
| 157.5 | & 158.4 | & 375.6 | & 205.9 | & 207.8 | & 2.7 | & 3.5 | & 1.4 | & 1.3 | RC100 | | | |

& 8 & 660. 1 & 503. 5 & 504. 1 & 254. 1 & 168. 7 & 169. 5 & 351. 9 & 227. 4
 & 229. 2 & 2. 6 & 3. 0 & 1. 4 & 1. 4 midruleextbf{ALL} & extbf{87} &
 extbf{392. 4} & extbf{163. 5} & extbf{163. 8} & extbf{121. 3} & extbf{44.
 8} & extbf{45. 1} & extbf{187. 1} & extbf{57. 1} & extbf{57. 8} & extbf{3.
 2} & extbf{3. 6} & extbf{1. 5} & extbf{1. 3} ottomruleend{tabular}
 %label{vrptw-table-average}end{adjustwidth}end{table}

%paragraph{Adding k -best columns to the RMP}Since the subproblem solver is able to provide the k -best solutions at each iteration, we have run a second set of experiments. For each column generation method, we have solved each instance with k equal to \$10\$, \$50\$, \$100\$, \$200\$ and \$300\$. In Table [ef{vrptw-table-relative-k_columns}](#) we show the results of these experiments where column k denotes the maximum number of columns added at each iteration to the RMP. For the three classes with 25 customers (C25, R25 and RC25), the SCG and the PDCGM have a similar overall performance up to a point in which the SCG outperforms the PDCGM. This is due to the fact that the RMPs are solved more efficiently by the solver in CPLEX than HOPDM. However, it is important to note that all the instances in these classes, are solved in less than 1 second in average by the two strategies. Now, if we take into account classes with 50 and 100 customers, the results obtained considering the total CPU time show that the PDCGM is consistently more efficient than the SCG for every k . In terms of column generation iterations, the results obtained with the PDCGM and SCG for different values of k when 50 customers are considered are similar and there is not a clear winner. However, if larger instances are considered (i. e., 100 customers), the PDCGM outperforms the SCG. On the

other hand, the PDCGM is consistently better than the ACCPM in both performance measures, for all the classes and for all the choices of k . Note that the performance of the ACCPM seems to be unaffected by the number of columns added per iteration. For all the strategies and values of k , the PDCGM with $k = 200$ is the most efficient setting on average, as it is 1.6\$ and 4.5\$ times faster than the best results obtained with the SCG ($k=300$) and the ACCPM ($k=100$), respectively.

Average results on 87\$ instances of the VRPTW for the SCG, PDCGM and ACCPM adding up to \$k\$ columns at a time.

| | SCG | PDCGM | ACCPM | SCG/PDCGM | ACCPM/PDCGM |
|------|-------|-------|-------|-----------|-------------|
| C25 | 35.8 | 0.4 | 0.4 | 18.9 | 0.1 |
| R25 | 19.3 | 0.1 | 0.1 | 22.9 | 0.1 |
| RC25 | 25.6 | 0.4 | 0.4 | 99.1 | 0.9 |
| C50 | 101.1 | 9.4 | 9.5 | 28.1 | 1.8 |
| R50 | 49.8 | 3.9 | 3.9 | 40.1 | 2.0 |
| RC50 | 53.4 | 5.8 | 5.9 | 45.0 | 3.8 |
| C100 | 271.3 | 9.3 | 9.6 | 1.2 | 1.5 |

9 & 93. 5 & 94. 3 & 49. 3 & 16. 5 & 16. 9 & 91. 7 & 32. 0 & 32. 5 & 5. 5 & 5. 6
 & 1. 9 & 1. 9 & R100 & 152. 4 & 125. 4 & 125. 8 & 86. 2 & 52. 0 & 52. 7 &
 205. 5 & 105. 2 & 106. 9 & 1. 8 & 2. 4 & 2. 4 & 2. 0 & RC100 & 147. 8 & 118.
 3 & 118. 7 & 78. 3 & 52. 0 & 52. 6 & 207. 1 & 135. 7 & 137. 1 & 1. 9 & 2. 3 &
 2. 6 & 2. 6 \midrule & extbf{ALL} & extbf{93. 7} & extbf{40. 0} & extbf{40.
 2} & extbf{44. 2} & extbf{14. 5} & extbf{14. 8} & extbf{128. 7} & &
 extbf{32. 2} & extbf{32. 8} & extbf{2. 1} & extbf{2. 7} & extbf{2. 9} & &
 extbf{2. 2} \midrule multirow{10}{*}{extbf{50}} & C25 & 20. 0 & 0. 3 & 0.
 3 & 17. 8 & 0. 1 & 0. 2 & 43. 1 & 0. 2 & 0. 3 & 1. 1 & 1. 4 & 2. 4 & 1. 5 & R25
 & 10. 2 & 0. 1 & 0. 1 & 17. 3 & 0. 1 & 0. 1 & 126. 8 & 0. 1 & 0. 3 & 0. 6 & 0. 6
 & 7. 4 & 2. 2 & RC25 & 14. 5 & 0. 3 & 0. 3 & 19. 6 & 0. 2 & 0. 3 & 95. 4 & 0. 9
 & 1. 1 & 0. 7 & 0. 8 & 4. 9 & 3. 3 & C50 & 49. 6 & 5. 2 & 5. 3 & 23. 0 & 1. 4 &
 1. 6 & 55. 1 & 3. 1 & 3. 3 & 2. 2 & 3. 3 & 2. 4 & 2. 1 & R50 & 24. 3 & 2. 4 & 2.
 4 & 26. 7 & 1. 3 & 1. 5 & 155. 8 & 4. 3 & 4. 7 & 0. 9 & 1. 5 & 5. 8 & 3. 0 &
 RC50 & 27. 8 & 3. 2 & 3. 2 & 29. 8 & 2. 3 & 2. 5 & 139. 3 & 9. 4 & 9. 8 & 0. 9
 & 1. 3 & 4. 7 & 3. 8 & C100 & 131. 0 & 46. 8 & 47. 7 & 40. 6 & 12. 9 & 13. 6
 & 88. 4 & 31. 0 & 32. 0 & 3. 2 & 3. 5 & 2. 2 & 2. 4 & R100 & 68. 2 & 58. 8 &
 59. 3 & 55. 3 & 32. 3 & 33. 6 & 195. 9 & 95. 1 & 97. 9 & 1. 2 & 1. 8 & 3. 5 &
 2. 9 & RC100 & 69. 5 & 58. 0 & 58. 4 & 47. 5 & 31. 4 & 32. 2 & 205. 4 & 133.
 6 & 135. 4 & 1. 5 & 1. 8 & 4. 3 & 4. 2 \midrule & extbf{ALL} & extbf{45. 2} &
 extbf{19. 5} & extbf{19. 7} & extbf{31. 0} & extbf{9. 3} & extbf{9. 7} & &
 extbf{125. 8} & extbf{30. 5} & extbf{31. 3} & extbf{1. 5} & extbf{2. 0} & &
 extbf{4. 1} & extbf{3. 2} \midrule multirow{10}{*}{extbf{100}} & C25 &
 16. 6 & 0. 2 & 0. 2 & 16. 6 & 0. 1 & 0. 2 & 43. 9 & 0. 2 & 0. 4 & 1. 0 & 1. 2 &
 2. 7 & 1. 8 & R25 & 8. 8 & 0. 1 & 0. 1 & 15. 8 & 0. 1 & 0. 2 & 126. 8 & 0. 1 &

0. 3 & 0. 6 & 0. 5 & 8. 1 & 2. 1 & RC25 & 12. 3 & 0. 2 & 0. 2 & 18. 4 & 0. 2 &
 0. 3 & 95. 1 & 0. 9 & 1. 1 & 0. 7 & 0. 7 & 5. 2 & 3. 3 & C50 & 39. 3 & 4. 4 & 4.
 5 & 21. 0 & 1. 5 & 1. 8 & 55. 6 & 3. 2 & 3. 6 & 1. 9 & 2. 4 & 2. 6 & 2. 0 & R50
 & 18. 5 & 2. 0 & 2. 0 & 23. 4 & 1. 1 & 1. 4 & 154. 5 & 4. 2 & 4. 8 & 0. 8 & 1. 4
 & 6. 6 & 3. 5 & RC50 & 22. 3 & 2. 8 & 2. 8 & 25. 3 & 1. 8 & 2. 1 & 139. 5 & 9.
 5 & 10. 0 & 0. 9 & 1. 4 & 5. 5 & 4. 9 & C100 & 94. 7 & 35. 2 & 36. 3 & 31. 8 &
 11. 8 & 12. 8 & 90. 9 & 31. 1 & 32. 9 & 3. 0 & 2. 8 & 2. 9 & 2. 6 & R100 & 53.
 1 & 45. 7 & 46. 2 & 40. 0 & 23. 3 & 24. 6 & 196. 8 & 94. 2 & 98. 2 & 1. 3 & 1.
 9 & 4. 9 & 4. 0 & RC100 & 50. 9 & 43. 0 & 43. 4 & 41. 6 & 25. 3 & 26. 5 &
 204. 4 & 130. 2 & 132. 7 & 1. 2 & 1. 6 & 4. 9 & 5. 0 midrule& extbf{ALL} &
 extbf{34. 5} & extbf{14. 9} & extbf{15. 2} & extbf{25. 9} & extbf{7. 3} &
 extbf{7. 8} & extbf{126. 0} & extbf{30. 1} & extbf{31. 3} & extbf{1. 3} &
 extbf{1. 9} & extbf{4. 9} & extbf{4. 0} midrulemultirow{10}{*}
 {extbf{200}} & C25 & 13. 6 & 0. 2 & 0. 2 & 15. 9 & 0. 1 & 0. 3 & 44. 7 & 0. 2
 & 0. 4 & 0. 9 & 0. 8 & 2. 8 & 1. 6 & R25 & 7. 0 & 0. 1 & 0. 1 & 15. 1 & 0. 1 &
 0. 2 & 126. 8 & 0. 2 & 0. 3 & 0. 5 & 0. 4 & 8. 4 & 1. 8 & RC25 & 9. 9 & 0. 2 &
 0. 2 & 18. 0 & 0. 2 & 0. 4 & 94. 4 & 0. 9 & 1. 1 & 0. 5 & 0. 5 & 5. 2 & 2. 8 &
 C50 & 31. 1 & 3. 5 & 3. 6 & 19. 3 & 1. 1 & 1. 6 & 57. 6 & 3. 0 & 3. 8 & 1. 6 &
 2. 3 & 3. 0 & 2. 4 & R50 & 14. 8 & 1. 6 & 1. 7 & 21. 5 & 1. 0 & 1. 4 & 155. 6 &
 4. 2 & 4. 9 & 0. 7 & 1. 2 & 7. 2 & 3. 5 & RC50 & 18. 1 & 2. 1 & 2. 1 & 22. 6 &
 1. 6 & 2. 1 & 140. 0 & 9. 6 & 10. 2 & 0. 8 & 1. 0 & 6. 2 & 4. 9 & C100 & 69. 3
 & 27. 0 & 28. 2 & 27. 9 & 9. 7 & 11. 2 & 92. 7 & 31. 0 & 34. 9 & 2. 5 & 2. 5 &
 3. 3 & 3. 1 & R100 & 41. 4 & 36. 3 & 36. 9 & 35. 0 & 19. 5 & 21. 5 & 198. 8 &
 90. 3 & 100. 8 & 1. 2 & 1. 7 & 5. 7 & 4. 7 & RC100 & 41. 6 & 35. 4 & 35. 9 &
 33. 5 & 21. 1 & 22. 7 & 204. 3 & 127. 5 & 132. 8 & 1. 2 & 1. 6 & 6. 1 & 5. 8

midrule& extbf{ALL} & extbf{26. 9} & extbf{11. 9} & extbf{12. 1} &
 extbf{23. 2} & extbf{6. 1} & extbf{6. 9} & extbf{126. 9} & extbf{29. 3} &
 extbf{31. 9} & extbf{1. 2} & extbf{1. 8} & extbf{5. 5} & extbf{4. 7}
 midrulemultirow{10}{*}{extbf{300}} & C25 & 12. 3 & 0. 2 & 0. 2 & 16. 3 &
 0. 1 & 0. 4 & 46. 1 & 0. 2 & 0. 5 & 0. 8 & 0. 5 & 2. 8 & 1. 3 & R25 & 6. 6 & 0.
 1 & 0. 1 & 14. 3 & 0. 1 & 0. 2 & 126. 8 & 0. 1 & 0. 3 & 0. 5 & 0. 3 & 8. 8 & 1.
 6 & RC25 & 10. 3 & 0. 2 & 0. 2 & 17. 4 & 0. 2 & 0. 5 & 96. 0 & 0. 9 & 1. 2 & 0.
 6 & 0. 4 & 5. 5 & 2. 5 & C50 & 26. 3 & 3. 0 & 3. 2 & 19. 1 & 1. 2 & 2. 0 & 58.
 6 & 2. 9 & 4. 1 & 1. 4 & 1. 6 & 3. 1 & 2. 0 & R50 & 12. 8 & 1. 5 & 1. 5 & 20. 6
 & 0. 9 & 1. 4 & 155. 5 & 4. 2 & 5. 1 & 0. 6 & 1. 1 & 7. 6 & 3. 6 & RC50 & 16. 4
 & 2. 1 & 2. 1 & 21. 8 & 1. 6 & 2. 2 & 140. 0 & 9. 2 & 9. 9 & 0. 8 & 1. 0 & 6. 4
 & 4. 5 & C100 & 57. 0 & 24. 2 & 25. 8 & 28. 0 & 9. 5 & 11. 9 & 96. 6 & 31. 2
 & 38. 6 & 2. 0 & 2. 2 & 3. 4 & 3. 2 & R100 & 35. 7 & 30. 8 & 31. 5 & 33. 4 &
 18. 7 & 21. 7 & 197. 6 & 85. 4 & 100. 3 & 1. 1 & 1. 5 & 5. 9 & 4. 6 & RC100 &
 36. 9 & 30. 7 & 31. 3 & 32. 6 & 20. 0 & 22. 3 & 206. 4 & 124. 6 & 132. 0 & 1.
 1 & 1. 4 & 6. 3 & 5. 9 midrule& extbf{ALL} & extbf{23. 3} & extbf{10. 3} &
 extbf{10. 7} & extbf{22. 6} & extbf{5. 8} & extbf{7. 0} & extbf{127. 7} &
 extbf{28. 3} & extbf{32. 2} & extbf{1. 0} & extbf{1. 5} & extbf{5. 7} &
 extbf{4. 6} ottomruleend{tabular}%label{vrptw-table-relative-

k_columns}end{adjustwidth}end{table}%Additionally, we have tested the
 three described column generation strategies in more challenging instances
 with \$200\$, \$400\$ and \$600\$ customers, which were proposed in
 cite{homerberger2005}. Table ef{vrptw-table-relative-large} shows the
 results of this third round of experiments, adding \$300\$ columns per
 iteration. The columns have the same meaning as in Table ef{vrptw-table-

average}. For all these instances, the PDCGM requires less CPU time and fewer iterations when compared with the SCG and the ACCPM. For the most difficult instance the PDCGM is \$2.1\$ and \$6.4\$ times faster than the SCG and the ACCPM, respectively.

Results on large instances of the VRPTW for the SCG, PDCGM and ACCPM adding 300 columns at a

time.

| | SCG | PDCGM | ACCPM | SCG/PDCGM | ACCPM/PDCGM |
|--------|-------|-------|-------|-----------|-------------|
| C200 | 85 | 33 | 41 | 29 | 13 |
| R200 | 15 | 169 | 72 | 82 | 2.9 |
| RC200 | 2.7 | 5.8 | 5.4 | 2.7 | 5.8 |
| C400 | 57 | 36 | 43 | 45 | 26 |
| R400 | 34 | 423 | 192 | 202 | 1.3 |
| RC400 | 1.2 | 9.4 | 5.9 | 67 | 105 |
| C600 | 110 | 57 | 77 | 88 | 385 |
| R600 | 567 | 607 | 1.2 | 1.2 | 6.8 |
| RC600 | 6.9 | 137 | 453 | 552 | 53 |
| C800 | 171 | 186 | 272 | 886 | 909 |
| R800 | 2.6 | 3.0 | 5.1 | 4.9 | 865 |
| RC800 | 84 | 596 | 640 | 636 | 2994 |
| C1000 | 3076 | 1.6 | 1.4 | 7.6 | 4.8 |
| R1000 | 189 | 2706 | 2789 | 113 | 1360 |
| RC1000 | 1436 | 521 | 6548 | 6649 | 1.7 |
| C1200 | 1.9 | 4.6 | 4.6 | 183 | 1921 |
| R1200 | 2335 | 48 | 496 | 510 | 482 |
| RC1200 | 5115 | 5173 | 3.8 | 4.6 | 10.0 |
| C1400 | 10.1 | 222 | 7226 | 7558 | 118 |
| R1400 | 4142 | 4260 | 897 | 25599 | 25870 |
| RC1400 | 1.9 | 1.8 | 7.6 | 6.1 | 258 |
| C1600 | 18701 | 18972 | 150 | 8677 | 8844 |
| R1600 | 923 | 56177 | 56683 | 1.7 | 2.1 |
| RC1600 | 6.2 | 6.4 | 6.2 | 6.4 | 6.4 |

relative-large}end{adjustwidth}end{table}%Finally, the benefits of using the PDCGM for solving the relaxation of the VRPTW after applying DWD are accentuated with the size and difficulty of the instances, so the larger the instance, the larger the benefits of using this column generation strategy.

subsection{Capacitated lot-sizing problem with setup times}In order to cover a wider spectrum of applications, we have considered the capacitated lot-sizing problem with setup times described in Chapter ef{ch: formulations}. The problem decomposes naturally in blocks for different items and therefore m different subproblems are obtained (one per item). This allows to test the column generation strategies when m essentially different columns are added in a disaggregated framework which differs from the previous two applications. Each subproblem is a single-item lot sizing problem with modified production and setup costs, and without capacity constraints. Hence, it can be solved by the Wagner-Whitin algorithm cite{WagWhi58}. We have selected 751 instances proposed in cite{TriThoMcC89} to test the aforementioned column generation strategies. The SCG and the PDCGM approaches are initialized using a single-column Big- M technique. The coefficients of this column are set to 0 in the capacity constraints and 1 in the convexity constraints. In the ACCPM approach, after several settings, we have chosen $u^{\{0\}} = 10.0e$ as the initial dual point. For all the column generation strategies we use the same subproblem solver which is our own implementation of the Wagner-Whitin algorithm cite{WagWhi58}. For each column generation strategy, we found that the 751 instances were solved in less than 100 seconds. The majority of them were solved in less than 0.1 seconds. From these

results, no meaningful comparisons and conclusions can be derived, so we have modified the instances in order to challenge the column generation approaches. For each instance and for each product we have replicated their demands 5 times and divided the capacity, processing time, setup time and costs by the same factor. Also, we have increased the capacity by 10% . Note that we have increased the size of the problems in time periods but not in items and therefore, all instances remain feasible. In Table [ef{clspst-table-average}](#), we show a summary of our findings using these modified instances. We have grouped the instances in seven different classes. Small instances are included in classes E, F and W while classes G, X1, X2 and X3 contain larger instances. Instances in classes E and F contain 6 items and 75 time periods while instances in class W have 4 or 8 items and 75 time periods. In class G, the instances have 6 , 12 or 24 items and 75 or 150 time periods. Classes X1, X2 and X3 contain instances with 100 time periods and 10 , 20 and 30 items, respectively. For each class and strategy we present: the number of column generation iterations (ite), the average CPU time required to solve the subproblems in seconds (or(s)) and the average total CPU time required for the column generation (tot(s)) in seconds. Column $\#$ indicates the number of instances per class. The last row (ALL) shows the average results considering the 751 modified instances. From Table [ef{clspst-table-average}](#), we can conclude that the strategies have different performances for the classes with small instances and on average each strategy requires less than 2 seconds to solve an instance from these classes. If we consider the total CPU time, the SCG is slightly better for classes E and F, and the

ACCPM outperforms the other two strategies only in class W. Considering the oracle times, we observe that for small instances the PDCGM outperforms the SCG due to the reduction in the number of outer iterations. However this reduction is somehow lost due to the fact that the PDCGM requires considerable time to solve the RMPs while the time required by the SCG is negligible. Now, if we observe the performance of the strategies on the classes with larger instances (emph{i. e.}, G, X1, X2 and X3), the PDCGM outperforms the other two strategies on average in both performance measures. Furthermore, considering the \$751\$ instances, the PDCGM reduces the average number of outer iterations and total CPU time when compared with the ACCPM and the SCG.

Average results on \$751\$ instances of the CLSPST for the SCG, PDCGM and ACCPM adding one column per subproblem at a time.

| | SCG | PDCGM | ACCPM | SCG/PDCGM | ACCPM/PDCGM |
|----|------|-------|-------|-----------|-------------|
| E | 58 | 38.1 | 0.7 | 0.7 | 0.5 |
| F | 70 | 33.4 | 0.6 | 0.6 | 0.9 |
| G | 71 | 44.8 | 6.6 | 6.6 | 0.6 |
| X1 | 40.4 | 0.7 | 0.9 | 1.2 | 0.8 |
| X2 | 48.6 | 0.8 | 1.1 | 1.2 | 0.7 |
| X3 | 66.4 | 1.2 | 1.2 | 55.3 | 1.0 |
| W | 12 | 66.4 | 1.2 | 1.2 | 55.3 |

& 4. 7 & 43. 2 & 5. 2 & 5. 6 & 1. 4 & 1. 4 & 1. 3 & 1. 2 X1 & 180 & 47. 5 & 4. 2 & 4. 2 & 28. 8 & 2. 4 & 3. 0 & 35. 2 & 3. 0 & 3. 3 & 1. 7 & 1. 4 & 1. 2 & 1. 1 X2 & 180 & 42. 6 & 7. 4 & 7. 5 & 20. 5 & 3. 5 & 3. 9 & 27. 4 & 4. 6 & 5. 0 & 2. 1 & 1. 9 & 1. 3 & 1. 3 X3 & 180 & 48. 9 & 12. 7 & 12. 8 & 18. 7 & 4. 7 & 5. 2 & 24. 3 & 6. 1 & 6. 7 & 2. 6 & 2. 5 & 1. 3 & 1. 3 midruleextbf{ALL} & extbf{751} & extbf{44. 7} & extbf{6. 6} & extbf{6. 6} & extbf{25. 1} & extbf{3. 0} & extbf{3. 5} & extbf{32. 4} & extbf{3. 9} & extbf{4. 3} & extbf{1. 8} & extbf{1. 9} & extbf{1. 3} & extbf{1. 2}

ottomruleend{tabular}%egin{tablenotes}item [1] A subset of \$7\$ instances could not be solved by the ACCPM using the default accuracy level, $\delta = 10^{-6}$ (\$4\$ from class X2 and \$3\$ from class X3). To overcome this we have used $\delta = 10^{-$

5}\$end{tablenotes}end{threeparttable}label{clspst-table-

average}end{adjustwidth}end{table}In addition to the previous

experiment, we have created a set of more challenging instances. We have

taken \$3\$ instances from cite{TriThoMcC89}, which were used in

cite{degraeve2007} as a comparison set. Additionally, we have selected \$8\$

instances from the sets of larger classes, X2 and X3. This small set of \$11\$

instances (emph{i. e.,} G30, G53, G57, X21117A, X21117B, X21118A,

X21118B, X31117A, X31117B, X31118A, X31118B) has been replicated \$5\$,

\$10\$, \$15\$ and \$20\$ times following the same procedure described above.

The summary of our findings is presented in Table ef{clspst-table-

replicated_11}, where column \$r\$ denotes the factor used to replicate the

selected instances. From the results, we see that for every choice of \$r\$, the

PDCGM requires fewer outer iterations and less CPU time on average, when

compared with the ACCPM and the SCG. Considering the 44 instances (11 instances and 4 values for r), the PDCGM is 2.8 and 2.6 times faster than the SCG and the ACCPM, respectively.

[H]caption{Average results on 11 modified instances of the CLSPST for the SCG, PDCGM and ACCPM adding one column per subproblem at a

time.}setlength{abcolsep}{4pt}scriptsizeegin{adjustwidth}{-1in}{-

1in}centeringegin{tabular}{cccccccccccc}oprule& multicolumn{3}{c}

{\extbf{SCG}} & multicolumn{3}{c}{\extbf{PDCGM}} & multicolumn{3}{c}

{\extbf{ACCPM}} & multicolumn{2}{c}{inyextbf{SCG/PDCGM}} &

multicolumn{2}{c}{inyextbf{ACCPM/PDCGM}}midruleextbf{r} & extbf{ite}

& extbf{or(s)} & extbf{tot(s)} & extbf{ite} & extbf{or(s)} & extbf{tot(s)} &

extbf{ite} & extbf{or(s)} & extbf{tot(s)} & extbf{ite} & extbf{tot(s)} &

extbf{ite} & extbf{tot(s)} midrule5 & 27.5 & 4.7 & 4.7 & 11.5 & 1.5 & 1.

6 & 22.5 & 3.1 & 3.2 & 2.4 & 3.0 & 2.0 & 2.1 10 & 32.0 & 62.7 & 62.7

& 15.6 & 20.4 & 21.0 & 29.5 & 49.1 & 49.5 & 2.0 & 3.0 & 1.9 & 2.4 15

& 38.4 & 308.8 & 308.8 & 20.0 & 103.8 & 106.2 & 36.4 & 273.2 & 274.

3 & 1.9 & 2.9 & 1.8 & 2.6 20 & 45.5 & 975.6 & 975.8 & 25.9 & 350.5 &

358.4 & 42.4 & 938.7 & 941.0 & 1.8 & 2.7 & 1.6 & 2.6

midruleextbf{ALL} & extbf{35.8} & extbf{337.9} & extbf{338.0} &

extbf{18.3} & extbf{119.0} & extbf{121.8} & extbf{32.7} & extbf{316.

0} & extbf{317.0} & extbf{2.0} & extbf{2.8} & extbf{1.8} & extbf{2.6}

ottomruleend{tabular}%label{clspst-table-

replicated_11}end{adjustwidth}end{table}%If we consider the average CPU

time per iteration for the CLSPST modified instances, the PDCGM is the most efficient among the studied strategies, while the SCG and the ACCPM have

very similar times per iteration. From the evidence gathered so far, one could infer that for some applications taking optimality and stabilization strategies as separated objectives may not originate any saving. However, if one can combine both objectives the resulting method can produce important savings in terms of CPU time and column generation iterations.

subsection{Performance profiles for large instances}label{subsec:

perf_prof}In order to complement our numerical comparisons, we have included performance profiles cite{dolanANDmore2002} for each application considering large instances. In short, performance profiles provide information about the behaviour of different methods for a given metric when solving a set of instances. In our case we are interested in two type of metrics. When comparing the PDCGM against the SCG and the ACCPM, we are interested in the number of outer iterations (calls to the subproblem(s)) and the total CPU time needed to solve an instance. For a better understanding of performance profiles, we will briefly describe the methodology proposed in cite{dolanANDmore2002}. Having the result for a particular metric (emph{e. g.,} total CPU time or outer iterations) obtained by using different methods, let us define \mathcal{M} and \mathcal{I} as the set of methods and instances, respectively. Then, for every i in \mathcal{I} and m in \mathcal{M} , we define $t_{i, m}$ as the result of the metric when solving instance i with method m . The baseline for every instance will be given by the best result obtained by any of the methods. In our case, and for all metrics considered, this is the minimum of the values among all methods. Therefore, the performance ratio can be defined as:

$$r_{i, m} = \frac{t_{i, m}}{\min_{m' \in \mathcal{M}} t_{i, m'}}$$

$\{\minlimits_{k \in \mathcal{M}} \{t_{i, k}\}\}$, for all $i \in \mathcal{I}$, for all $m \in \mathcal{M}$. end{equation} Additionally, if we define

$\mathcal{S}_m(\alpha) = \{i \in \mathcal{I} : r_{i, m} \leq \alpha\}$, then the cumulative distribution function of method m for the performance ratio is $h_m(\alpha) = \frac{1}{|\mathcal{I}|} |\mathcal{S}_m(\alpha)|$, for all $m \in \mathcal{M}$.

where $h_m(\alpha)$ represents the probability that the result of method m is between a ratio α with respect to the best result among all methods. Note that this type of analysis can cope with

solver/method failures so it gives a good measure of robustness. Then, in order to generate the cumulative distribution plots of every method for a set of instances (performance profiles), we set several values of α (x -axis) and plot them against the corresponding $h_m(\alpha)$ (y -axis). The figures we present in the next section were created with an slightly modified MATLAB script available at [url{http://www. mcs. anl. gov/~more/cops/}](http://www.mcs.anl.gov/~more/cops/). Having the results for the aforementioned applications in terms of average and classified by number of columns added per iteration give us a good idea of the efficiency of the methods. It shows clearly that the larger the instances, the better the overall performance of the PDCGM. In order to mitigate the influence of poor performances of some of the strategies in very specific instances (large CPU times or number of outer iterations), we also present the performance profile results for all the difficult instances and all the column strategies. We have selected only instances which challenge the column generation for every application.

subsubsection{Cutting stock problem} In Figures [ef{fig: pp_csp: subfig1}](#) and [ef{fig: pp_csp: subfig2}](#) we have the performance profiles in terms of outer

iterations and total CPU time, respectively. We have considered instances in class U and the 14 large instances presented in Table ef{csp-table-very_large} and all values of k previously considered. These results clearly complement the discussion in the previous section. It clarifies any doubt and shows that the PDCGM is the best method to solve large instances for this class of problems. Independently of the number of columns added per iteration and instances considered, the PDCGM always requires fewer outer iterations than the SCG and the ACCPM as shown in Figure ef{fig: pp_csp: subfig1}. Also, in terms of CPU time the PDCGM is the most efficient technique for the vast majority of larger instances and when it is not the best, it does not perform badly (only at a factor less than 1.5 from the best strategy). One can observe that the ACCPM is not very competitive and that more than 50% of the instances require at least 6 times more CPU time to be solved than the best result obtained with either the PDCGM or the SCG.

```

5in}centeringsubfigure[Outer iterations]{includegraphics[keepaspectratio=
true, clip= true, viewport = 85 105 760 465, scale = 0. 60]
{figures/outerCSP}label{fig: pp_csp: subfig1}

}
subfigure[Total CPU time]{includegraphics[keepaspectratio= true, clip=
true, viewport = 85 105 760 460, scale = 0. 60] {figures/timeCSP}label{fig:
pp_csp: subfig2}

```

```

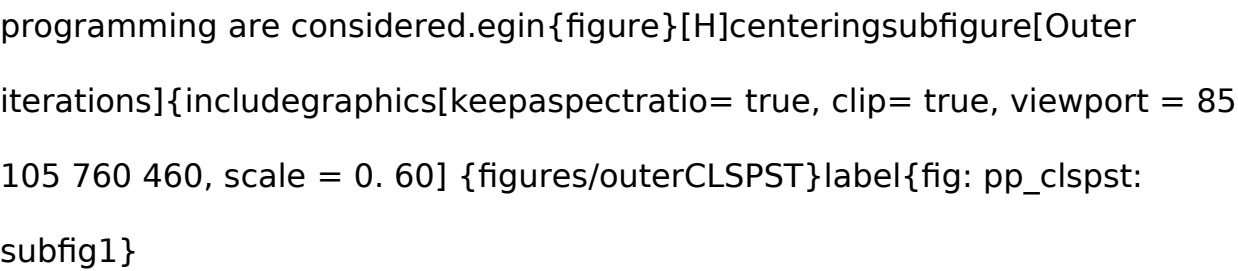
}
caption{Performance profiles for the CSP with the SCG, PDCGM and ACCPM
(large instances)}label{fig: pp_csp}
%end{adjustwidth}end{figure}subsubsection{Vehicle routing problem with
time windows}For the VRPTW, we have considered instances in classes
C100, R100 and RC100 plus $9$ large instances with $200$, $400$ and
$600$ customers (see Table ef{vrptw-table-relative-large}).egin{figure}
[H]centeringsubfigure[Outer iterations]{includegraphics[keepaspectratio=
true, clip= true, viewport = 85 105 760 460, scale = 0. 60]
{figures/outerVRPTW}}label{fig: pp_vrptw: subfig1}

}
subfigure[Total CPU time]{includegraphics[keepaspectratio= true, clip=
true, viewport = 85 105 760 460, scale = 0. 60]
{figures/timeVRPTW}}label{fig: pp_vrptw: subfig2}

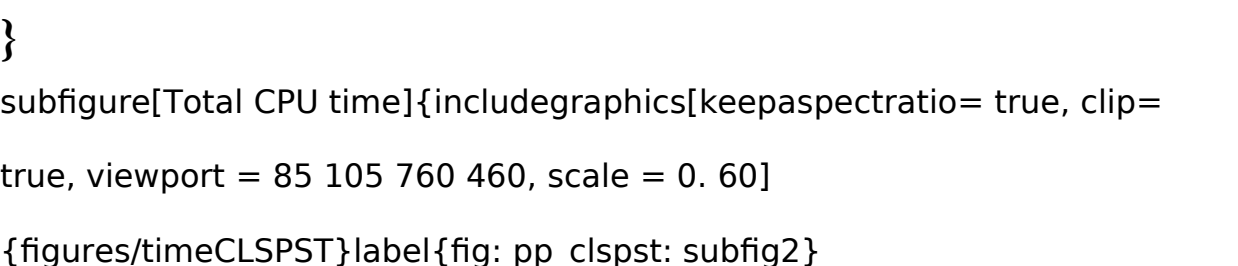
}
caption{Performance profiles for the VRPTW with the SCG, PDCGM and
ACCPM (large instances)}label{fig: pp_vrptw}end{figure}In Figures ef{fig:
pp_vrptw: subfig1} and ef{fig: pp_vrptw: subfig2} we present the
performance profiles for outer iterations and total CPU time, respectively
obtained by the three column generation strategies. Similar to the results for
the CSP considering large instances, the PDCGM is the most efficient method
among the three approaches considered. It performs consistently better than
the SCG and ACCPM in more than $90\%$ of the instances in both
performance measures. subsubsection{Capacitated lot-sizing problem with

```

setup times} For the performance profiles of the CLSPST, we have considered instances in classes G, X1, X2 and X3. Additionally, we have included the replicated instances with r equal to 10, 15 and 20. In Figures [ef{fig: pp_clspst: subfig1}](#) and [ef{fig: pp_clspst: subfig2}](#) we have the performance profiles for outer iterations and total CPU time, respectively. Again, PDCGM is the strategy that obtains the best results for most of the selected instances. It is the most efficient strategy in terms of CPU time in more than 75\% of the cases and when it is not the best, it does not perform poorly compared to the other two strategies. Differently to the results with other two applications, namely the CSP and VRPTW, the ACCPM performs much better than the SCG and it offers the best performance in terms of CPU time in almost 20\% of the instances. With these additional comparisons we aim to provide the reader with enough evidence to support our conclusion that the PDCGM is %consistently the best column generation strategythe variant with the best overall performance among the ones considered in this study when large instances in the context of integer programming are considered.



[ef{figure}\[H\]centeringsubfigure\[Outer iterations\]{includegraphics\[keepaspectratio= true, clip= true, viewport = 85 105 760 460, scale = 0. 60\] {figures/outerCLSPST}}label{fig: pp_clspst: subfig1}](#)



[subfigure\[Total CPU time\]{includegraphics\[keepaspectratio= true, clip= true, viewport = 85 105 760 460, scale = 0. 60\] {figures/timeCLSPST}}label{fig: pp_clspst: subfig2}](#)

}

caption{Performance profiles for the CLSPST with the SCG, PDCGM and ACCPM (large instances)}label{fig:

pp_clspst}end{figure}subsection{Additional comments about stabilized column generation}Although no computational study has been performed considering an artificially stabilized column generation based on simplex, we would like to refer to two papers which provide us with some notion on how much can be gained by stabilizing the applications considered in this thesis. In cite{BriLemMeuMicPerVan08}, the authors present a comprehensive computational study comparing the standard column generation (Kelley method) against the bundle method, a stabilized cutting plane method which uses quadratic stabilization terms as shown in Section ef{sec: cg: strategies}. A set of five different applications have been considered in the computational experiments, including the CSP and the CLSPST (MILS problem in their paper). Regarding the CSP, the results in that paper indicate that using the bundle method may slightly reduce the number of column generation iterations at the cost of worsening the total CPU time by a factor greater than \$3\$ (see Tables 1 and 2 in cite{BriLemMeuMicPerVan08}). Furthermore, in the results obtained for the CLSPST, we observe that the bundle method behaves poorly in terms of the number of outer iterations and CPU time when compared with the standard column generation (see Table 12 in cite{BriLemMeuMicPerVan08}). In Rousseau emph{et al.} cite{Rousseau2007}, the authors propose an interior point column generation technique in which the dual solutions are convex combinations of extreme dual points of the RMP that are obtained by randomly modifying the

dual objective function. To analyse the computational performance of their approach, they have used the set of large VRPTW instances with 100 customers, namely instances included in classes C100, R100 and RC100. Only the results of 22 out of 29 instances were presented by the authors. Their comparison involves the implementations of the standard column generation as well as a stabilized version called BoxPen technique (duMerle1999). Since a different subproblem solver, another version of CPLEX and a different machine have been used in their computational experiments, it would not be appropriated to make a straightforward comparison of the figures presented in their tables with those presented in Tables ef{vrptw-table-average} and ef{vrptw-table-relative-k_columns}. Hence, we have considered the gains obtained by each approach in relation to the standard column generation. According to their results, a well-tuned implementation of the BoxPen stabilization reduces the number of outer iterations by 16%, on average, while the total difference regarding CPU time is negligible when compared with the standard column generation. The interior point stabilization (IPS) proposed by the authors shows a better performance than the BoxPen stabilization, being 1.38 times faster than the standard column generation technique. For the same set of instances (22 out of 29), the PDCGM is 2.03 times faster than the SCG on average ($k = 200$). Two final comments with regard to the ACCPM are needed. Looking at our computational results and considering all the applications studied in this thesis, it seems that the ACCPM suffers when multiple columns are added at each iteration of the column generation. As mentioned in one of the very few papers in the context of column generation

and Lagrangian relaxation for integer programming problems using the ACCPM, the starting point is critical for the success of the algorithm cite{BelTadVia06}. Initializing the ACCPM with poor columns may originate unnecessary iterations at the beginning of the column generation, which is expensive. This may be emphasised by the addition of unnecessary columns at every iteration making the method converging slowly. A remedy to this could be to carefully choose the initial point for each instance. However, this is impractical and out of the scope of this study. Also, in order to guarantee a good performance of the ACCPM so it can efficiently reoptimize after new columns are added, the theory requires some safeguards. For instance, if multiple columns are added in one iteration and the old analytic centre deeply violates these new constraints in the dual space, theoretically the ACCPM struggles and no warmstarting is possible cite{goffinvial1999, goffinvial2000}. As a final remark of this chapter, from our results one can observe that unlike the SCG, the strategies based on interior point methods, namely the PDCGM and the ACCPM, spend a non-negligible amount of time solving the RMPs. This is because reoptimizing interior point methods is not as straightforward as reoptimizing with simplex-type methods. This issue will be extensively discussed in the next chapter.