Comput Optim Appl (2008) 41: 185–204
DOI 10.1007/s10589-008-9207-4

Progress in the dual simplex algorithm for solving large scale LP problems: techniques for a fast and stable implementation

Achim Koberstein

Received: 30 March 2006 / Revised: 14 September 2006 / Published online: 16 September 2008

Abstract During the last fifteen years the dual simplex method has become a strong contender in solving large scale LP problems. However, the lack of descriptions of important implementation details in the research literature has led to a great performance gap between open-source research codes and commercial LP-systems. In this paper we present the mathematical algorithms, computational techniques and implementation details, which are the key factors for our dual simplex code to close this gap. We describe how to exploit hyper-sparsity in the dual simplex algorithm. Furthermore, we give a conceptual integration of Harris' ratio test, bound flipping and cost shifting techniques and describe a sophisticated and efficient implementation. We also address important issues of the implementation of dual steepest edge pricing. Finally we show on a large set of practical large scale LP problems, that our dual simplex code outperforms the best existing open-source and research codes and is competitive to the leading commercial LP-systems on our most difficult test problems.

Keywords Linear programming · Dual simplex algorithm · MIP-solver MOPS

1 Introduction

While the primal simplex algorithm has been in the center of research interest for decades and subject of countless publications, this has not been the case regarding its dual counterpart. After Lemke [20] had presented the dual simplex method in 1954, it was not considered to be a competitive alternative to the primal simplex method for nearly forty years. This eventually changed in the early 1990s mainly due to the contributions of Forrest and Goldfarb [9] and Fourer [10].

A. Koberstein (✉)
Decision Support & Operations Research Lab, University of Paderborn, Warburger Str. 100, 33100 Paderborn, Germany
e-mail: koberstein@dsor.de

During the last decade commercial solvers made great progress in establishing the dual simplex method as a general solver for large-scale LP problems. Nowadays, large scale LP problems can be solved either by an interior point, primal simplex or dual simplex algorithm or a combination of such algorithms. In fact, extensive computational studies indicate, that the performance of the dual simplex is typically superior to that of the primal simplex algorithm (cf. [2]). In practice, there are often LP-models, for which one of the three methods clearly outperforms the others. Furthermore, the dual simplex method plays an important role for solving mixed-integer linear programming problems (MIPs) by branch-and-bound type approaches.

Despite of its success and relevance for future research only few publications in research literature explicitly discuss implementation details of mathematical or computational techniques proposed for the dual simplex algorithm. The lack of descriptions of implementation details in the research literature has led to a great performance gap between open-source research codes[1] and commercial LP-systems, which is frequently documented in independent benchmarks of optimization software (cf. [26]).

In this paper we strive to fill this gap and present computational techniques and implementation details, which we consider key for a high performance dual simplex code. Our work is based on the MIP-solver *Mathematical OPtimization System* (*MOPS*) (see [30]), which has been deployed in many practical applications for almost two decades and belongs to the few competitive systems in the world to solve large-scale linear and mixed integer programming problems.

The remainder of this paper is structured as follows. At first we describe an elaborated version of the dual simplex algorithm for general LPs in Sect. 2. Computational techniques and implementation issues for each part of the algorithm are discussed in Sect. 3. Here, we give a brief description of how to exploit hyper-sparsity based on the ideas of Gilbert and Peierls [11]. Furthermore, we present a conceptual integration of Harris' ratio test [15], bound flipping [10, 24] and cost shifting techniques [12] and describe a sophisticated and efficient implementation. We also address important issues of the implementation of dual steepest edge pricing [9]. In Sect. 4 we finally show on a large set of practical large scale LP problems, that our dual simplex code outperforms the best existing open-source and research codes and is competitive to the leading commercial LP-systems on our most difficult test problems.

2 The elaborated dual simplex method for general linear programming problems

We consider linear programming problems (LPs) of the form

$$\begin{aligned} \min \quad & \mathbf{c}^T\mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} \qquad (1) \\ & \mathbf{l} \le \mathbf{x} \le \mathbf{u}, \end{aligned}$$

[1] An exception is the LP code, which is being developed in the COIN open-source initiative [21]. However, this code is largely undocumented and no research papers have yet been published about its internals.

where $\mathbf{c}, \mathbf{x} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{l}, \mathbf{u} \in (\mathbb{R} \cup \{-\infty, +\infty\})^n$ and $\mathbf{A} \in \mathbb{R}^{m \times n}$ with $rank(\mathbf{A}) = m$, $m, n \in \mathbb{N}$ and $m < n$. In the following we will denote by $\mathcal{J} = \{1, \ldots, n\}$ the set of column indices. We call a variable x_j with $j \in \mathcal{J}$ *free* if $l_j = -\infty$ and $u_j = +\infty$. We call it *boxed* if $l_j > -\infty$ and $u_j < +\infty$. If $l_j = u_j = a$ for some $a \in \mathbb{R}$ we call it *fixed*.

Common LP-systems usually allow LPs of even more general form (featuring *ranges* instead of a right-hand-side vector $\mathbf{b}$) and transform them into the form (1) in a standardized way by adding slack and surplus variables. In MOPS $\mathbf{A}$ assumes the form $[\bar{\mathbf{A}}, \mathbf{I}]$, where $\mathbf{I}$ is the identity matrix, and $\mathbf{b}$ is set to zero. The variables associated with $\bar{\mathbf{A}}$ are called *structural variables* (short: *structurals*), those associated with $\mathbf{I}$ are called *logical variables* (short: *logicals*). Ranges in the *external model representation* (*EMR*) (which corresponds to the user's model) transform to bounds on the logical variables in the *internal model representation* (*IMR*).

A basis $\mathcal{B} = \{k_1, \ldots, k_m\}$ is an ordered subset of $\mathcal{J}$, such that the submatrix $\mathbf{B} = \mathbf{A}_{\mathcal{B}}$ is nonsingular. The set of nonbasic column indices is denoted by $\mathcal{N} = \mathcal{J} \setminus \mathcal{B}$. A primal basic solution for a basis $\mathcal{B}$ is constructed by setting every primal nonbasic variable x_j, $j \in \mathcal{N}$, to one of its finite bounds (or to zero if it is free) and computing the primal basic variables as $\mathbf{x}_{\mathcal{B}} = \mathbf{B}^{-1}(\mathbf{b} - \mathbf{A}_{\mathcal{N}}\mathbf{x}_{\mathcal{N}})$. $\mathcal{B}$ is called primal feasible if all of the primal basic variables are within their bounds, i.e., $l_j \le x_j \le u_j$ for all $j \in \mathcal{B}$. $\mathcal{B}$ is called dual feasible if, for the reduced cost vector $\mathbf{d} = \mathbf{c} - \mathbf{c}_{\mathcal{B}}^T\mathbf{B}^{-1}\mathbf{A}$, $d_j \ge 0$ if $x_j = l_j$ and $d_j \le 0$ if $x_j = u_j$ for all $j \in \mathcal{N}$. The reduced cost value of a nonbasic free primal variable can only be dual feasible if it is zero. The reduced cost value of a nonbasic fixed primal variable is always dual feasible. A dual infeasible reduced cost value of a nonbasic boxed primal variable can always be made dual feasible by setting the variable to its other bound. Clearly, if such a bound switch is performed, the basic primal variables have to be updated.

The dual simplex method starts with a dual feasible basis and keeps changing the basis until primal feasibility is achieved or the problem turns out to be dual unbounded. A computationally appropriate description of the method is given in Algorithms 1, 2, 3 and 4. It basically corresponds to the revised dual simplex algorithm for general LPs, enhanced by dual steepest edge pricing and a generalized dual ratio test. Due to these extensions we call it *Elaborated Dual Simplex Algorithm*. In this paper we assume, that a dual feasible basic solution has been obtained by a dual phase 1 method (see [18] for an comparison of suitable dual phase 1 methods).

In step 1 of Algorithm 1, an LU-factorization of the basis matrix B is built, primal and dual basic solution vectors are computed, and the vector of dual steepest edge weights is initialized.

In step 2, which we call dual pricing, we determine a primal infeasible basic variable with index $p \in \mathcal{B}$ and $p = k_r$ which leaves the basis. Here we use dual steepest edge pricing (DSE) based on the method called "Dual Algorithm I" by Forrest and Goldfarb [9]. The vector of DSE weights is denoted by $\boldsymbol{\beta}$. The efficiency and actual performance of DSE pricing depends heavily on implementation details (see Sect. 3.3).

In steps 3 and 4 we compute the transformed pivot row $\boldsymbol{\alpha}^r$, which is required to determine an entering variable with index $q \in \mathcal{N}$ in step 5. Instead of the standard dual ratio test we use a generalized version, which we call *bound flipping ratio test*

Algorithm 1: Dual simplex method with dual steepest edge pricing, bound flipping ratio test and update of **d**.

```
Input: LP in computational form (1), dual feasible basis
        $\mathcal{B} = \{k_1, \dots, k_m\}$, primal nonbasic solution vector $\mathbf{x}_\mathcal{N}$.
Output: Optimal basis $\mathcal{B}$ with optimal primal basic solution $\mathbf{x}$ or
        proof that LP is dual unbounded (=primal infeasible).
(Step 1) Initialization
    Compute an LU-factorization of $\mathbf{B}$.
    Compute $\tilde{\mathbf{b}} = \mathbf{b} - \mathbf{A}_\mathcal{N}\mathbf{x}_\mathcal{N}$ and solve $\mathbf{B}\mathbf{x}_\mathcal{B} = \tilde{\mathbf{b}}$ for $\mathbf{x}_\mathcal{B}$.
    Solve $\mathbf{B}^T\mathbf{y} = \mathbf{c}_\mathcal{B}$ for $\mathbf{y}$ and compute $\mathbf{d}_\mathcal{N} = \mathbf{c}_\mathcal{N} - \mathbf{A}_\mathcal{N}^T\mathbf{y}$.
    Initialize $\boldsymbol{\beta}$ (set $\boldsymbol{\beta}^T \leftarrow (1, \dots, 1)^T$ if $\mathbf{B} = \mathbf{I}$).
    Compute $Z = \mathbf{c}^T\mathbf{x}$.
(Step 2) Pricing
    If $\mathbf{l}_\mathcal{B} \le \mathbf{x}_\mathcal{B} \le \mathbf{u}_\mathcal{B}$ then terminate: $\mathcal{B}$ is an optimal basis.
    Else select $r \in \arg\max_{i\in\{1,\dots,m\}}\{\frac{|\delta_i|^2}{\beta_i} : \delta_i \neq 0\}$ with $\delta_i = x_{k_i} - l_{k_i}$ if
    $x_{k_i} < l_{k_i}$, $\delta_i = x_{k_i} - u_{k_i}$ if $x_{k_i} > u_{k_i}$ and $\delta_i = 0$ o.w. Set $p \leftarrow k_r$
    and $\delta \leftarrow \delta_r$.
(Step 3) BTran
    Solve $\mathbf{B}^T\boldsymbol{\rho}_r = \mathbf{e}_r$ for $\boldsymbol{\rho}_r$.
(Step 4) Pivot row
    Compute $\boldsymbol{\alpha}^r = \mathbf{A}_\mathcal{N}^T\boldsymbol{\rho}_r$.
(Step 5) Ratio Test
    Determine $q$ by Algorithm 2.
(Step 6) FTran
    Solve $\mathbf{B}\boldsymbol{\alpha}_q = \mathbf{a}_q$ for $\boldsymbol{\alpha}_q$.
(Step 7) DSE FTran
    Solve $\mathbf{B}\boldsymbol{\tau} = \boldsymbol{\rho}_r$ for $\boldsymbol{\tau}$.
(Step 8) Basis change and update
    Update $\boldsymbol{\beta}$. Set $\beta_i \leftarrow \beta_i - 2\frac{\alpha_q^i}{\alpha_q^r}\tau_i + (\frac{\alpha_q^i}{\alpha_q^r})^2\beta_r$ for all $i \neq r$ and
    $\beta_r \leftarrow (\frac{1}{\alpha_q^r})^2\beta_r$.
    Update $\mathbf{d}$ by Algorithm 3.
    Compute $\theta^P = \frac{\delta}{\alpha_q^p}$ and set $\mathbf{x}_\mathcal{B} \leftarrow \mathbf{x}_\mathcal{B} - \theta^P\boldsymbol{\alpha}_q$ and $x_q \leftarrow x_q + \theta^P$.
    Set $\mathcal{B} \leftarrow (\mathcal{B} \setminus \{p\}) \cup \{q\}$ and $\mathcal{N} \leftarrow (\mathcal{N} \setminus \{q\}) \cup \{p\}$.
    Update the LU-factorization of $\mathbf{B}$.
    Flip bounds in $\mathbf{x}_\mathcal{N}$ by Algorithm 4.
    Set $Z \leftarrow Z + \theta^D\delta$.
    Go to step 1.
```

Algorithm 2: Selection of q with the BRFT.

```
If $x_p < l_p$ set $\tilde{\boldsymbol{\alpha}}^r \leftarrow -\boldsymbol{\alpha}^r$, if $x_p > u_p$ set $\tilde{\boldsymbol{\alpha}}^r \leftarrow \boldsymbol{\alpha}^r$.
Let $\mathcal{Q} \leftarrow \{j : j \in \mathcal{N}, x_j$ free or $(x_j = l_j$ and $\tilde{\alpha}^r_j > 0)$ or
                                        $(x_j = u_j$ and $\tilde{\alpha}^r_j < 0)\}$.
while $\mathcal{Q} \neq \emptyset$ and $\delta \geq 0$ do
    Select $q \in \arg\min_{j \in \mathcal{Q}}\{\frac{d_j}{\tilde{\alpha}^r_j}\}$.
    Set $\delta \leftarrow \delta - (u_q - l_q)|\alpha^r_q|$.
    Set $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{q\}$.
end
If $\mathcal{Q} = \emptyset$ then terminate: the LP is dual unbounded.
Set $\theta^D \leftarrow \frac{d_q}{\alpha^r_q}$.
```

Algorithm 3: Update of $\mathbf{d}$ and $\mathbf{x}_\mathcal{B}$ for the BRFT.

```
Set $\tilde{\mathbf{a}} \leftarrow 0$, $\mathcal{T} \leftarrow \emptyset$ and $\Delta Z \leftarrow 0$.
forall $j \in \mathcal{N}$ do
    Set $d_j \leftarrow d_j - \theta^D \alpha^r_j$.
    if $l_j \neq u_j$ then
        if $x_j = l_j$ and $d_j < 0$ then
            Set $\mathcal{T} \leftarrow \mathcal{T} \cup \{j\}$.
            Set $\tilde{\mathbf{a}} \leftarrow \tilde{\mathbf{a}} + (u_j - l_j)\mathbf{a}_j$.
            Set $\Delta Z \leftarrow \Delta Z + (u_j - l_j)c_j$.
        else if $x_j = u_j$ and $d_j > 0$ then
            Set $\mathcal{T} \leftarrow \mathcal{T} \cup \{j\}$.
            Set $\tilde{\mathbf{a}} \leftarrow \tilde{\mathbf{a}} + (l_j - u_j)\mathbf{a}_j$.
            Set $\Delta Z \leftarrow \Delta Z + (l_j - u_j)c_j$.
        end
    end
end
Set $d_p \leftarrow -\theta^D$.
if $\mathcal{T} \neq \emptyset$ then
    Solve $\mathbf{B}\Delta\mathbf{x}_\mathcal{B} = \tilde{\mathbf{a}}$ for $\Delta\mathbf{x}_\mathcal{B}$.
    Set $\mathbf{x}_\mathcal{B} \leftarrow \mathbf{x}_\mathcal{B} - \Delta\mathbf{x}_\mathcal{B}$.
    Set $\Delta Z \leftarrow \Delta Z - \sum_{j \in \mathcal{B}} c_j \Delta x_j$.
end
Set $Z \leftarrow Z + \Delta Z$.
```

(*BFRT*) (cf. [10, 16, 19] and [22, 24]). The bound flipping ratio test is one significant reason why the dual simplex method typically outperforms the primal simplex

Algorithm 4: Update of $\mathbf{x}_{\mathcal{N}}$ for the BRFT.

```
forall $j \in \mathcal{T}$ do
    if $x_j = l_j$ then
        Set $x_j \leftarrow u_j$.
    else
        Set $x_j \leftarrow l_j$.
    end
end
```

method.[2] It is based on the observation that the reduced cost value of a boxed nonbasic primal variable can be kept dual feasible even if it switches sign by setting the variable it to its opposite bound. This means that the dual step length can be further increased and breakpoints associated with boxed primal variables can be skipped as long as the dual objective function keeps improving. It can be seen that the rate of improvement decreases with every bound flip. When it drops below zero no further improvement can be made and the entering variable can be selected from the current set of bounding breakpoints (the set of breakpoints is denoted by $\mathcal{Q}$). If bound flips occur the primal basic variables have to be updated accordingly. To perform this update efficiently an additional FTran-type linear system has to be solved.

There are different ways to embed the bound flipping ratio test into the standard revised dual simplex method. In both Fourer's and Maros' description the selection of the entering variable and the update operations for the primal basic variables are integrated in the ratio test in the sense, that sets $\mathcal{T}^+$ and $\mathcal{T}^-$ of necessary bound flips as well as the summation of the corresponding columns of $\mathbf{A}$ in $\tilde{\mathbf{a}}$ are done in the main selection loop for q. Below, we present a version, where the update operations are strictly separated from the selection of the entering variable. The bound flips are not recorded in the ratio test, but during the update of the reduced cost vector $\mathbf{d}$. If an updated entry d_j is detected to be dual infeasible, the corresponding nonbasic primal variable (which must be boxed in this case) is marked for a bound flip and column $\mathbf{a}_j$ is added to $\tilde{\mathbf{a}}$. After the update of $\mathbf{d}$, a system $\mathbf{B}\Delta\mathbf{x}_{\mathcal{B}} = \tilde{\mathbf{a}}$ is solved for $\Delta\mathbf{x}_{\mathcal{B}}$ and $\mathbf{x}_{\mathcal{B}}$ is set to $\mathbf{x}_{\mathcal{B}} - \Delta\mathbf{x}_{\mathcal{B}}$. The actual bound flips in $\mathbf{x}_{\mathcal{N}}$ are performed at the very end of a major iteration. The three parts selection of the entering variable, update of $\mathbf{d}$ and $\mathbf{x}_{\mathcal{B}}$ and update of $\mathbf{x}_{\mathcal{N}}$ are described in Algorithms 2, 3 and 4, respectively. An efficient implementation of the BFRT is very important for the overall performance. Furthermore, the BFRT has to be combined with further computational techniques such as feasibility tolerances and cost shifting. These issues will be addressed in detail in Sect. 3.4.

In steps 6 and 7 of Algorithm 1 we solve two FTran-type linear systems to compute the transformed pivot column $\boldsymbol{\alpha}_q$ and the vector $\boldsymbol{\tau}$, which are used to update the primal basic variables and the DSE weights in step 8, respectively. The reduced costs, the objective function value, the basis and the LU-factorization are also updated in

[2]The closest to an analogous technique is the L_1 ratio test in the primal phase 1 (see e.g. [23]).

this step. For the FTran, BTran and LU-Update operations we use improved versions of the routines described in [32] (see Sect. 3.2).

3 Computational techniques and implementation

3.1 Data structures and computation of the pivot row

In the following, *dense storage* means that a simple array is used to store an exact image of a numerical vector, including zero entries. *Indexed storage* uses a second array for the indices of the nonzero elements in addition to a dense array for zero and nonzero values. In *packed storage* only nonzero entries and their indices are stored, both in compact form. The constraint matrix $\bar{\mathbf{A}}$ of the IMR is stored in columnwise compact form. $\mathbf{c}$, $\mathbf{l}$ and $\mathbf{u}$ are stored in simple dense arrays. Although at the beginning cost coefficients of logical variables are zero, the corresponding array has to be of size n to handle cost shiftings and perturbations. At the start a copy of the cost vector is created, which is used at the end of the method to restore the original cost vector after potential cost modifications. The current primal and dual basic solutions are stored in dense arrays as well.

For some models the computation of the pivot row $\boldsymbol{\alpha}^r$ (step 1 of Algorithm 1) can take more than 50% of the total solution time even for sophisticated implementations. In [3] Bixby et al. recommend row-wise instead of columnwise computation to be able to exploit sparsity in $\boldsymbol{\rho}_r$. While the computational effort can be drastically reduced especially for hyper-sparse models, row-wise computation also has some disadvantages as well. The need for an additional row-wise datastructure for $\bar{\mathbf{A}}$ significantly increases the memory requirements of the method. Furthermore, positions of $\boldsymbol{\alpha}^r$ corresponding to basic or fixed variables cannot easily be skipped. Bixby et al. recommend to maintain a row-wise data structure only for nonbasic nonfixed positions and update it in every basis change. Inserting a column into the data structure is usually very cheap, since the time required per nonzero element is constant. But for deleting a column this time is linear in the number of nonzeros of the affected row, so this operation can be quite expensive. In fact, in our tests we had some poor results for models with many more columns than rows. On the other hand we could not see a significant speedup compared to the version without updating, so we decided to use a complete row-wise storage of $\bar{\mathbf{A}}$ in our implementation. When the dual simplex method is used as part of a branch-and-cut procedure for mixed-integer programming this data structure has to be derived anyway for other purposes. A further disadvantage of the row-wise computation is that the build-up of an index stack requires an additional pass over the nonzero elements, since $\boldsymbol{\alpha}^r$ has to be held in a dense array. For these reasons, we use both the columnwise *and* the row-wise computation dependent on the density of $\boldsymbol{\rho}_r$. If it exceeds 30% then the columnwise computation is used. For models with many more columns than rows this threshold is lowered to 10%.

We use packed storage for the structural part of $\boldsymbol{\alpha}^r$ and $\boldsymbol{\rho}_r$ (which coincides with the logical part of $\boldsymbol{\alpha}^r$). While creating the packed storage causes some additional costs after the row-wise computation of $\boldsymbol{\alpha}^r$, its usage pays off in the dual ratio test and the update of $\mathbf{d}_{\mathcal{N}}$ if $\boldsymbol{\alpha}^r$ and $\boldsymbol{\rho}_r$ are sparse. This is particularly the case for large hyper-sparse problems. While $\boldsymbol{\rho}_r$ is usually also sparse on typical medium sized and small problems, the structural part of $\boldsymbol{\alpha}^r$ can get fairly dense. For these problems, on which the density of $\boldsymbol{\alpha}^r$ often exceeds 40%, dense storage would be advantageous. To avoid further complication of the code, we decided to pass on this option for now.

The input and output vectors of our FTran and BTran routines are all in indexed storage. Consequently, we need a loop over the nonzero elements of $\boldsymbol{\rho}_r$ after the BTran operation in step 3 to create the packed format. However, we also keep the dense representation of $\boldsymbol{\rho}_r$. It is needed as an input for the FTran operation in the context of the update of the DSE weights and for the columnwise computation of $\boldsymbol{\alpha}^r$. For the transformed pivot column $\boldsymbol{\alpha}_q$ we use indexed storage, which is readily obtained by the FTran operation in step 7. Depending on the density of $\boldsymbol{\alpha}_q$ we decide whether we use the corresponding index array during the update of $\mathbf{x}_{\mathcal{B}}$ and the DSE weights, which are stored in a dense array. If it exceeds 30%, we use dense processing.[3]

While the packed arrays are overwritten in every iteration, the dense arrays for $\boldsymbol{\rho}_r$ and $\boldsymbol{\alpha}_q$ have to be zero at the beginning of every iteration. We use the index arrays to zero out the dense arrays after each iteration, which is crucial especially for hyper-sparse problems (the careful organization of this clearing operation alone saved about 15% of runtime on the large hyper-sparse NetLib model ken-18).

3.2 Exploiting hyper-sparsity in FTran, BTran, LU-Update and factorization

Up to four linear systems have to be solved per iteration of the elaborated dual simplex method. Three of them are of the form $\mathbf{B}\boldsymbol{\alpha} = \mathbf{a}$ (**FTran**) and one is of the form $\mathbf{B}^T\boldsymbol{\pi} = \mathbf{h}$ (**BTran**). As in virtually every competitive LP-code our solution procedures are based on an LU-factorization of the basis matrix $\mathbf{B}$, which is updated at the end of each iteration. Algorithms and implementation for LU-factorization and update are described in [31, 32] and [17]. In an arbitrary simplex iteration we have an LU-factorization of the form:

$$\tilde{\mathbf{L}}^k \cdots \tilde{\mathbf{L}}^{s+1}\tilde{\mathbf{L}}^s \cdots \tilde{\mathbf{L}}^1\mathbf{B} = \tilde{\mathbf{U}}, \tag{2}$$

where $\tilde{\mathbf{L}}^j$ $(1 \le j \le s)$ are permuted lower triangular column-eta-matrices stemming from the LU-factorization, $\tilde{\mathbf{L}}^j$ $(s+1 \le j \le k)$ are permuted row-eta-matrices stemming from the LU-update and $\tilde{\mathbf{U}} = \mathbf{P}^{-1}\mathbf{U}\mathbf{P}^{-1}$ is a permuted upper triangular matrix. Based on (2) we decompose the FTran operation into three parts:

FTranL-F: Compute $\bar{\bar{\boldsymbol{\alpha}}} = \tilde{\mathbf{L}}^s \cdots \tilde{\mathbf{L}}^1\mathbf{a}$. (3a)

FTranL-U: Compute $\bar{\boldsymbol{\alpha}} = \tilde{\mathbf{L}}^k \cdots \tilde{\mathbf{L}}^{s+1}\bar{\bar{\boldsymbol{\alpha}}}$. (3b)

FTranU: Solve $\mathbf{P}^{-1}\mathbf{U}\mathbf{P}\boldsymbol{\alpha} = \bar{\boldsymbol{\alpha}}$ for $\boldsymbol{\alpha}$. (3c)

[3] This entails that the code for the respective update routines practically doubles: one version loops over the index array, the other version loops over m.

Similarly, the BTran operation is decomposed into three parts

BTranU: Solve $\mathbf{P}^{-1}\mathbf{U}^T\mathbf{P}\bar{\boldsymbol{\pi}} = \mathbf{h}$ for $\bar{\boldsymbol{\pi}}$. (4a)

BTranL-U: Compute $\bar{\bar{\boldsymbol{\pi}}} = (\tilde{\mathbf{L}}^{s+1})^T \cdots (\tilde{\mathbf{L}}^k)^T\bar{\boldsymbol{\pi}}$. (4b)

BTranL-F: Compute $\boldsymbol{\pi} = (\tilde{\mathbf{L}}^1)^T \cdots (\tilde{\mathbf{L}}^s)^T\bar{\bar{\boldsymbol{\pi}}}$. (4c)

In the late nineteen-nineties it was independently reported by Bixby [2, 4] and by Hall and McKinnon [14] that for some LP problems remarkable improvements in performance can be achieved by exploiting the sparsity of the result and intermediate result vectors of these systems even more aggressively. Hall and McKinnon introduced the term *hyper-sparsity* for this phenomenon. One way to exploit hyper-sparsity is to apply a technique published by J.R. Gilbert and T. Peierls [11] as early as 1988 in the linear algebra community. First in a *symbolical phase* a list and a feasible ordering of the nonzero positions in the result vector are determined. This is done by depth-first-search (DFS) on the compact representations of the involved matrices. In the subsequent *numerical phase* the standard sparsity exploiting solution methods are deployed based on this list. In our code the DFS is implemented iteratively. To achieve an optimal cache behavior the arrays, which are required to organize the DFS, should be arranged in straight succession in main memory. In [17] we give detailed pseudo-code for the hyper-sparse solution algorithms deployed in our code.

The hyper-sparse solution technique can be applied in those subsystems, in which sparsity in the result vectors can be exploited by skipping rows or columns of the system's matrix corresponding to zero entries. This can be done in FTranL-F, FTranU, BTranU and BTranL-U. For each of these systems we therefore provide three types of solution methods: one for dense processing (simple forward/backward substitution), one for sparse processing (row-wise/column-wise forward/backward substitution, skipping of rows/columns) and another one for hyper-sparse processing. Depending on a simple estimate for the density of the (intermediate) result vectors, which is based on the average number of resulting nonzeros in the same system over all previous iterations since the last refactorization, we dynamically switch between the three versions. According to our experience it is advisable to use the hyper-sparse version if the density falls below 5% and the dense version if the density exceeds 70%.

The computation of the row-eta-matrix $\tilde{\mathbf{L}}^j$ in the LU-update procedure is in general equivalent to a reduced BTranU operation. Therefore, the sparse and hyper-sparse solution techniques mentioned above can as well be applied in the LU-update procedure. In our tests we learned that the update of the permutation in the Suhl/Suhl LU-update can become a relatively time consuming operation on very hyper-sparse models (such as ken-18 from the NetLib test set [28]). In our code we split this operation in two separate loops, one for the update operation for $\mathbf{P}^{-1}$ and one for $\mathbf{P}$. Due to better data locality the first loop can be executed much faster than the second one. Alternatively, one could use a sophisticated implementation of the classical Forrest/Tomlin update for this type of problems, which we found in the COIN LP code [21]. There, the update of the permutation arrays is realized as a constant time operation. However, it is well known that the Forrest/Tomlin is generally inferior to the Suhl/Suhl update since it produces more fill-in. Therefore, we use the latter in our code. It is an open question to us, whether a constant time update of the perturbation can also be achieved for the Suhl/Suhl update.

3.3 Pricing

We learned from the work with our code that the DSE weights have to be initialized and updated with greatest possible accuracy. Even small errors can lead to a substantial increase of the iteration count. Therefore we use the following numerically favorable reformulation of the DSE update formula:

$$\bar{\beta}_i = \beta_i + \alpha_q^i \left(\alpha_q^i \bar{\beta}_r + \kappa \tau_i\right) \quad \text{with } \kappa = \frac{-2}{\alpha_q^r}. \tag{5}$$

As recommended in [9] we avoid negative weights by setting $\bar{\beta}_i = \max\{\bar{\beta}_i, 10^{-4}\}$. On most models (5) worked substantially better than the original version.

We tested several variants of initializing the weights as well. Computing the weights from their definition for an arbitrary basis is very expensive (up to m BTran operations). For an all-logical basis we know that 1.0 is the correct initial value. But even if we start with an all-logical basis, it may change substantially during the dual phase I, where we do not necessarily apply DSE pricing at all (e.g. in Pan's dual phase I, cf. [18]). Furthermore, due to the permutation of the basis after each refactorization, the vector of DSE weights becomes invalid. Therefore we just reset the weights to 1.0 after a refactorization in early versions of our code. Later, when we realized the importance of ensuring the highest possible accuracy of the weights, we changed this simple strategy as follows:

- If no crash procedure[4] is used we initialize the weights with 1.0. Otherwise we perform the expensive initialization from their definition (BTran is only necessary for rows, which are not unit vectors).
- Prior to each refactorization the weights are copied into a dense vector of length n in such a way that weight β_i is assigned to the position corresponding to the index k_i of the associated basic variable. After the factorization the weights are reassigned taking into account the new permutation of the basis.
- To have the correct values in dual phase II, the weights are always updated in dual phase I, even if they are not used (e.g. in Pan's phase I algorithm).
- During the update we save the old values of the weights that are changed in a packed array. If an iteration is aborted after the update, we use this array to restore the original values. This situation can occur if the stability test in LU-update fails.

With these improvements of accuracy of the DSE weights we achieved a great reduction of both total iteration count and runtime compared to the simpler implementation.

A crashed starting basis led to a smaller number of iterations only if the DSE weights were correctly initialized. However, even then total runtime did not improve

[4]See e.g. [25, p. 244ff] for an overview of suitable methods.

in most of the cases, since the average time per iteration increased due to more fill-in in the LU-factors. Therefore, by default we start with an all-logical basis. If the dual simplex method is deployed to reoptimize during branch-and-bound, there is no time to recompute the weights in every node of the B&B tree. Here, the default is to reuse the weights of the last LP-iteration.

In the dual pricing step we have to loop through the weighted primal infeasibilities to select the leaving variable. Typically, 20–40% of the primal basic variables are infeasible at the start. For some problems this fraction is even lower. During the solution process the number of infeasibilities usually decreases continuously, until the basis is finally primal feasible. To exploit this observation, we changed our code to maintain an explicit indexed vector of the primal infeasibilities.[5] In the following xpifs denotes the dense array of the squared infeasibilities and xpifsi denotes the corresponding index array. This data structure is rebuilt after each recomputation of the primal basic solution $\mathbf{x}_{\mathcal{B}}$, which is done after each refactorization of the basis matrix. In a normal iteration it is updated during the update of $\mathbf{x}_{\mathcal{B}}$. Four cases can occur during the update of a primal basic variable x_{k_i}:

1. x_{k_i} stays primal feasible. The vector of primal infeasibilities does not change.
2. x_{k_i} becomes primal infeasible. The squared infeasibility is recorded in xpifs and the index i is added to xpifsi, which length increases by one.
3. x_{k_i} becomes primal feasible. The i-th entry of xpifs is replaced by a small constant 10^{-50} to signal feasibility. xpifsi is not changed.
4. x_{k_i} stays primal infeasible. xpifs[i] is adjusted. xpifsi is not changed.

Typically, the number of indices stored in xpifsi increases moderately between two refactorizations and drops down to the actual number of infeasibilities after each recomputation of $\mathbf{x}_{\mathcal{B}}$. Considering only the entries of xpifsi in the pricing loop led to great improvements of runtime especially on large models. Considerable savings could be achieved on many smaller problems as well.

On some large easy problems the computational effort spent in the pricing loop could be further dramatically reduced by a partial pricing strategy. In each iteration we dynamically adjust the fraction of considered leaving candidates depending on the current relative iteration speed. As an estimate of the computational effort per iteration we take a ratio of the number of nonzero elements in the LU-factors and the number of rows. To give all candidates the same probability to be considered we determine a random starting point for the pricing loop. On some problems this randomization of pricing also lead to fewer degenerate iterations. Furthermore, it reduces the risk of numerical cycling.

3.4 Ratio test

The bound flipping ratio test described in Algorithm 2 already offers a much greater flexibility to avoid small pivots than the standard ratio test (see also [24]). Let

$$k \in \{1, \dots, |\mathcal{Q}|\} : \quad \delta_k \geq 0 \quad \text{and} \quad \delta_{k+1} < 0 \tag{6}$$

[5]Actually, the squared infeasibilities are stored to further speed up the DSE pricing loop.

be the last mini-iteration before the slope δ becomes negative. If $|\tilde{\alpha}^r_{q_k}|$ is too small and much smaller than the largest available pivot element, we can always go back to a preceding mini-iteration. The goal is the find a good compromise between a well sized pivot element and a good progress in the objective function. However, especially for problems with few boxed variables, the additional flexibility does not suffice. Also, it might sometimes be more efficient to accept small dual infeasibilities and achieve a lower iteration count by going the farthest possible step in the bound flipping ratio test. Therefore, we will now describe how the well known idea of feasibility tolerances introduced by P. Harris [15] can be incorporated into the BFRT.

We basically apply Harris' ratio test in every mini-iteration. Let $\mathcal{Q}^i$ be the set of remaining breakpoints in mini-iteration i and

$$\mathcal{Q}^i_l = \left\{ j \in \mathcal{Q}^i : \tilde{\alpha}^r_j > 0 \right\} \quad \text{and} \quad \mathcal{Q}^i_u = \left\{ j \in \mathcal{Q}^i : \tilde{\alpha}^r_j < 0 \right\}. \tag{7}$$

Let ϵ^D denote the dual feasibility tolerance (in our code: 10^{-7}). Then an upper bound Θ^i_{max} on the maximal step length in mini-iteration i can be determined by

$$\Theta^i_{max} = \min \left\{ \min_{j \in \mathcal{Q}^i_l} \left\{ \frac{d_j + \epsilon^D}{\tilde{\alpha}^r_j} \right\}, \min_{j \in \mathcal{Q}^i_u} \left\{ \frac{d_j - \epsilon^D}{\tilde{\alpha}^r_j} \right\} \right\}. \tag{8}$$

Instead of looking at only one breakpoint per mini-iteration we then consider all remaining breakpoints at once, which do not exceed the maximal step length Θ^i_{max}. We denote the set of these breakpoints by $\mathcal{K}_i$ with

$$\mathcal{K}_i = \left\{ j \in \mathcal{Q}^i : \frac{d_j}{\tilde{\alpha}^r_j} \le \Theta^i_{max} \right\}. \tag{9}$$

Finally, we can determine the possible entering index q_i and the corresponding dual step length by

$$q_i \in \arg\max_{j \in \mathcal{K}_i} \left\{ |\tilde{\alpha}^r_j| \right\} \quad \text{and} \quad \theta^D_i = \frac{d_{q_i}}{\tilde{\alpha}^r_{q_i}} \tag{10}$$

and perform the update operations

$$\mathcal{Q}_{i+1} = \mathcal{Q}_i \setminus \mathcal{K}_i \quad \text{and} \tag{11}$$

$$\delta_{i+1} = \delta_i - \sum_{j \in \mathcal{K}_i} (u_j - l_j) |\tilde{\alpha}^r_j|. \tag{12}$$

An important implementation issue is the choice of an adequate data structure for the set of breakpoints. The incorporation of Harris' ideas has significant consequences for this choice. Fourer [10] and Maros [24] recommend to organize the set of breakpoints as a heap data-structure to allow for a very fast access to the next smallest breakpoint. This implicates a considerable computational effort to build up the heap structure, which arises independently of the number of actually performed mini-iterations. We conducted some experiments, which approved Maros' observation, that the number of

Algorithm 5: Bound flipping ratio test with Harris' tolerance.

```
Phase 1: Determine candidate set.
If $x_p < l_p$ set $\tilde{\boldsymbol{\alpha}}^r \leftarrow -\boldsymbol{\alpha}^r$ and $\delta_0 \leftarrow l_p - x_p$.
If $x_p > u_p$ set $\tilde{\boldsymbol{\alpha}}^r \leftarrow \boldsymbol{\alpha}^r$ and $\delta_0 \leftarrow x_p - u_p$.
Compute $\mathcal{Q} \leftarrow \{j : j \in \mathcal{N}, x_j \textit{ free}$ or $(x_j = l_j$ and $\tilde{\alpha}^r_j > 0)$ or
   $(x_j = u_j$ and $\tilde{\alpha}^r_j < 0)\}$.
Compute $\Theta_{max} \leftarrow \min\{\min_{j\in\mathcal{Q}_l}\{\frac{d_j+\epsilon^D}{\tilde{\alpha}^r_j}\}, \min_{j\in\mathcal{Q}_u}\{\frac{d_j-\epsilon^D}{\tilde{\alpha}^r_j}\}\}$.

Phase 2: Reduce candidate set. Find interesting region for $\theta^D$.
Set $\theta^D \leftarrow 10.0 * \Theta_{max}$.
Set $\delta \leftarrow \delta_0$.
Set $\hat{\delta} \leftarrow 0$.
while $\delta - \hat{\delta} \geq 0$ do
    Set $\delta \leftarrow \delta - \hat{\delta}$.
    Compute $\tilde{\mathcal{Q}} \leftarrow \{j \in \mathcal{Q}_l : d_j - \theta^D\tilde{\alpha}^r_j < -\epsilon^D\} \cup \{j \in \mathcal{Q}_u : d_j - \theta^D\tilde{\alpha}^r_j > \epsilon^D\}$.
    Compute $\Theta_{max} \leftarrow \min\{\min_{j\in\mathcal{Q}_l}\{\frac{d_j+\epsilon^D}{\tilde{\alpha}^r_j}\}, \min_{j\in\mathcal{Q}_u}\{\frac{d_j-\epsilon^D}{\tilde{\alpha}^r_j}\}\}$.
    Set $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \tilde{\mathcal{Q}}$.
    Set $\hat{\delta} \leftarrow \sum_{j\in\tilde{\mathcal{Q}}}(u_j - l_j)|\tilde{\alpha}^r_j|$.
    Set $\theta^D \leftarrow 2.0 * \Theta_{max}$.
end

Phase 3: Perform BFRT with Harris' tolerance on interesting region.
while $\tilde{\mathcal{Q}} \neq \emptyset$ and $\delta \geq 0$ do
    Compute $\Theta_{max} \leftarrow \min\{\min_{j\in\tilde{\mathcal{Q}}_l}\{\frac{d_j+\epsilon^D}{\tilde{\alpha}^r_j}\}, \min_{j\in\tilde{\mathcal{Q}}_u}\{\frac{d_j-\epsilon^D}{\tilde{\alpha}^r_j}\}\}$.
    Let $\mathcal{K} = \{j \in \tilde{\mathcal{Q}} : \frac{d_j}{\tilde{\alpha}^r_j} \leq \Theta_{max}\}$.
    Select $q \in \arg\max_{j\in\mathcal{K}}\{|\tilde{\alpha}^r_j|\}$.
    Set $\tilde{\mathcal{Q}} \leftarrow \tilde{\mathcal{Q}} \setminus \mathcal{K}$.
    Set $\delta \leftarrow \delta - \sum_{j\in\mathcal{K}}(u_j - l_j)|\tilde{\alpha}^r_j|$.
end
If $\tilde{\mathcal{Q}} = \emptyset$ then terminate: the LP is dual unbounded.
Set $\theta^D \leftarrow \frac{d_q}{\alpha^r_q}$.
```

flipped variables may vary dramatically even in subsequent iterations. Furthermore, on instances with few or no boxed variables virtually no bound flips are possible at all. Therefore, we opted for an implementation based on simple linear search combined with a forceful reduction technique for the set of eligible candidate breakpoints.

Algorithm 5 describes our algorithmic realization of the BFRT with feasibility tolerance, which is inspired by the dual simplex implementation of the COIN LP code [21]. It proceeds in three phases. In phase 1, which needs one pass of the transformed pivot row, a complete set of breakpoints $\mathcal{Q}$ is determined. At the same time a Harris bound Θ_{max} is computed with respect to the first possible breakpoint. This bound is used as a starting point in phase 2, which tries to reduce the set of candidate breakpoints. This is done by doubling the dual step length and collecting passed breakpoints until the slope changes sign. Only for these breakpoints the actual BRFT+Harris procedure with its subsequent linear searches is conducted. Typically the number of candidates can be significantly reduced by very few passes in phase 2 and therefore, phase 3 can be carried out very efficiently. Further implementation details can be found in [17].

We also want to mention an interesting idea brought to our mind by P.Q. Pan in personal conversation, which could be named *partial ratio test*. In a first pass only those nonbasic nonfixed positions of the transformed pivot row are computed for which the corresponding reduced cost entry is zero. As soon as one of these positions defines a breakpoint (α_j^r nonzero and of right sign), it is taken as the entering variable. In this case the iteration must be degenerate and the further computation of the transformed pivot row as well as the dual ratio test and the update of the reduced cost vector can be skipped. Pan reports a significant improvement of his code by this idea. Up to now, we have not tested this technique, since it obviously collides with several other concepts in our code, as the bound flipping ratio test, the cost shifting scheme and the perturbation.

3.5 Shifting

Choosing a dual infeasible variable (within the tolerance) to enter the basis leads to a backward step and increases the risk of numerical cycling. The main approach to avoid backward steps and the creation of new dual infeasibilities is called *shifting* and was worked out in detail for the primal simplex method in [12] in the context of the anti-cycling procedure EXPAND. The idea is to locally manipulate the problem data to avoid the negative step. For the dual simplex method this means manipulations of cost coefficients. We use it to always guarantee a slightly positive dual step length and still fulfill the dual basic constraint for the entering variable, such that $\bar{d}_q = 0$ after the basis change. This can obviously make other nonbasic variables violate their feasibility bound after the update of the reduced costs. Therefore, in the EXPAND concept the positive "mini-step" is accompanied by a small increase of the feasibility tolerance in every iteration. The disadvantage of expanding the tolerance is though, that the new tolerance is valid for all of the nonbasic variables and thus allows for greater infeasibilities in the further course of the method. For this reason we chose an alternative way which is again inspired by the COIN code [21]. We perform further shifts only for those reduced cost values which would violate the feasibility tolerance.

Suppose that we have determined a dual step length $\theta^D = \frac{d_q}{\tilde{\alpha}_q^r} < 0$ at the end of the dual ratio test. Then the following is done:

1. Set $\theta^D \leftarrow 10^{-12}$.
2. Compute shift $\Delta_q \leftarrow \theta^D \tilde{\alpha}_q^r - d_q$.
3. Set $c_q \leftarrow c_q + \Delta_q$.
   Set $d_q \leftarrow d_q + \Delta_q$.

The additional shifts are computed as follows:

$$\Delta_j = \begin{cases} \max\left\{\theta^D \tilde{\alpha}_j^r - d_j - \epsilon^D, 0\right\} & \text{if } j \in \mathcal{Q}_l, \\ \min\left\{\theta^D \tilde{\alpha}_j^r - d_j + \epsilon^D, 0\right\} & \text{if } j \in \mathcal{Q}_u. \end{cases} \tag{13}$$

Since $\theta^D = 10^{-12}$ is very small, only few of these additional shifts are different from zero and among these most are in the range of 10^{-14} to 10^{-10}. That way the changes to the cost coefficients are kept as small as possible. We also experimented with random shifts of greater magnitude to achieve a more significant positive step, a greater distance to infeasibility and resolve degeneracy at the same time (a similar shifting variant was proposed by [33, p. 61] to break stalling and prevent cycling). However, our impression was that the COIN procedure is superior, if it is combined with the perturbation procedure described in the next section.

3.6 Perturbation

Our perturbation procedure, which is similar to the one implemented in the COIN LP code [21], is applied prior to the dual simplex method, if many cost coefficients have the same value. The determination of the cost changes is the main difference compared to other methods from the literature. While these methods aim solely at resolving degeneracy, our perturbation procedure also tries to exploit degeneracy to keep the number of nonzero elements in the basis matrix low. In general, the presence of degeneracy increases the flexibility in the dual ratio test. If the problem is numerically stable, this flexibility can be used to select entering variables with as few nonzero elements in their corresponding columns of the problem matrix as possible. The effect is, that the spike in the LU-update stays sparser and the number of nonzeros added to the LU-factorization lower. But if the problem is perturbed at the start, most of this flexibility is lost.[6] Therefore, the number of nonzero elements should be considered in the determination of the cost perturbations. In our code the determination of a cost perturbation ξ_j is performed in four steps:

1. Determine the right magnitude of ξ_j. Set ξ_j to a fixed value, which consists of a constant fraction (typically $100 \cdot \epsilon^D$) and a variable fraction dependent on the size of the cost coefficient (typically $\psi \cdot c_j$, with $\psi = 10^{-5}$).
2. Randomize ξ_j and set the right sign w.r.t. dual feasibility. This is done by setting

$$\xi_j \leftarrow \begin{cases} -0.5\xi_j(1+\mu) & \text{if } u_j < \infty, \\ 0.5\xi_j(1+\mu) & \text{o.w.,} \end{cases} \tag{14}$$

   where μ is a random number within the interval [0, 1]. Note, that by using (14) ξ_j basically keeps the size determined in step 1.
3. Incorporate nonzero counts. Dependent on the number ν_j of nonzero elements in column $\mathbf{a}_j$, ξ_j is multiplied by a weighting factor w_k:

$$\xi_j \leftarrow w_{\nu_j} \cdot \xi_j. \tag{15}$$

[6] In this sense the perturbation also works against Harris' ratio test.

The weight-vector **w** displays the tradeoff between the two goals "resolve degeneracy" and "keep nonzero count low". We use the following scheme:

$$\mathbf{w}^T = (10^{-2}, 10^{-1}, 1.0, 2.0, 5.0, 10.0, 20.0, 30.0, 40.0, 100.0). \tag{16}$$

If $\nu_j > 10$ we use $w_{10} = 100.0$ in (15). As mentioned before most practical LP problems have less than fifteen nonzero entries per column independent of the number of rows.

4. Ensure that ξ_j stays within an interval $[\xi_{min}, \xi_{max}]$ with

$$\xi_{min} = \min\{10^{-2}\epsilon^D, \psi\} \quad \text{and} \tag{17}$$

$$\xi_{max} = \max\left\{10^3\epsilon^D, \psi \cdot 10 \cdot \frac{1}{n}\sum_{j=1}^{n} c_j\right\}. \tag{18}$$

If ξ_j exceeds or falls below these thresholds, it is multiplied with 0.1 or 10, respectively, until it falls into the above interval.

At the end of the dual simplex method we restore the original cost vector, which can lead to a loss of dual feasibility of the current basic solution. If the method terminated due to primal feasibility, we switch to the primal simplex method (to be more precise: primal phase II) to restore dual feasibility (cf. [1]). Usually, only few iterations are necessary to find a new optimal basis. If the method terminated due to dual unboundedness, we just restore the cost vector and start over with the dual phase I if necessary to confirm unboundedness. The same procedure is used to remove the cost shiftings performed in the dual ratio test.

4 Numerical results

The computational results presented in this section are based on a set of one hundred test problems, which are taken from five different sources:

- 17 problems from the NetLib and Kennington test set (cf. [28]): CRE-B, CRE-D, D2Q06C, DEGEN4, DFL001, FIT2P, GREENBEA, GREENBEB, KEN-13, KEN-18, MAROS-R7, OSA-30, OSA-60, PDS-10, PDS-20, PILOT, PILOT87.
- 17 problems[7] from the MipLib2003 test set (cf. [27]): AIR04, ATLANTA-IP, DANO3MIP, DS, FAST0507, MOMENTUM2, MOMENTUM3, MSC98-IP, MZZV11, MZZV42Z, NET12, RD-RPLUSC-21, SEYMOUR, SP97AR, STP3D, T1717, VAN.
- 21 problems from the Mittelmann test set (cf. [26]): BAXTER, DBIC1, FOME11, FOME20, FXM4_6, GEN4, LP22, MOD2, NSCT2, PDS-100, PDS-40, QAP12, RAIL4284, RAIL507, SELF, SGPF5Y6, STORMG2_1000, STORMG2-125, WATSON_1, WATSON_2, WORLD.

[7] The LP-relaxations of these problems.

Table 1 Problem dimensions of five large problems from our test set

Name	Source	Structurals	Constraints	Nonzeros
MUN1_M_D	[8]	1479833	163142	3031285
MUN18_M_D	[8]	675333	62148	1379406
RAIL4284	[26]	1092610	4284	11279748
STP3D	[27]	204880	159488	662128
STORMG2_1000	[26]	1259121	528185	3341696

Table 2 Progress in our dual simplex code: implementation techniques in chronological order

Version	Added Implementation Technique	Time	Iters
1	First Version		
2	Dual Steepest Edge	−47%	−43%
3	Packed Storage of $\boldsymbol{\alpha}^r$	−17%	+1%
4	Packed Storage of $\boldsymbol{\rho}$	−7%	−3%
5	Vector of Primal Infeasibilities	−43%	−8%
6	Bound Flipping Ratio Test, Perturbation	−40%	−23%
7	Numerical Stability	−6%	−3%
8	Hyper-sparse FTran,BTran	−23%	−1%
9	Tight bounds after LP Preprocessing	−39%	+2%
10	Revised Dual Steepest Edge	−30%	−18%
11	Hyper-sparse LU-Update	−11%	0%
12	Randomized Partial Pricing	−15%	0%

- 20 problems from the BPMPD test set (cf. [5]): AA3, BAS1LP, CO9, CQ9, DBIR1, EX3STA1, JENDREC1, LPL1, LPL3, MODEL10, NEMSPMM2, NEMSWRLD, NUG08, NUG12, RAT5, SCFXM1-2R-256, SLPTSK, SOUTH31, T0331-4L, ULEVIMIN.
- 25 problems from our own collection of test problems (cf. [8]): FA, HAL_M_D, MUN1_M_D, MUN18_M_D, P01, P02, P03, P04, P05, P06, P07, P08, P09, P10, P11, P12, P13, P14, P15, P16, P17, P18, P19, P20, PTV15.

Table 1 shows problem dimensions of five large problems from our test set, which were the most difficult w.r.t. Cplex 9.1 runtime. For the other problems, dimensions and detailed individual benchmarking results can be found in [17].

Table 2 reflects the progress of a two-year development process. It shows the relative improvements in average CPU time and iteration count achieved by the different techniques discussed in this paper. With the first versions of our code we could not solve some of the NetLib test problems due to numerical problems. Therefore, these results are limited to a selection of thirteen large NetLib problems, which could already be solved with these early versions (CRE-B, CRE-D, D2Q06C, DFL001, FIT2D, FIT2P, KEN-11, KEN-13, KEN-18, MAROS-R7, OSA-14, OSA-30, OSA-60). The first version basically coincides with the revised dual simplex algorithm for general LPs with Devex pricing and simple ratio test. The two required systems of linear equations per iteration were solved based on the LU-factorization and -update described in [31] and [32]. Sparsity was exploited in FTran and BTran and by a row-

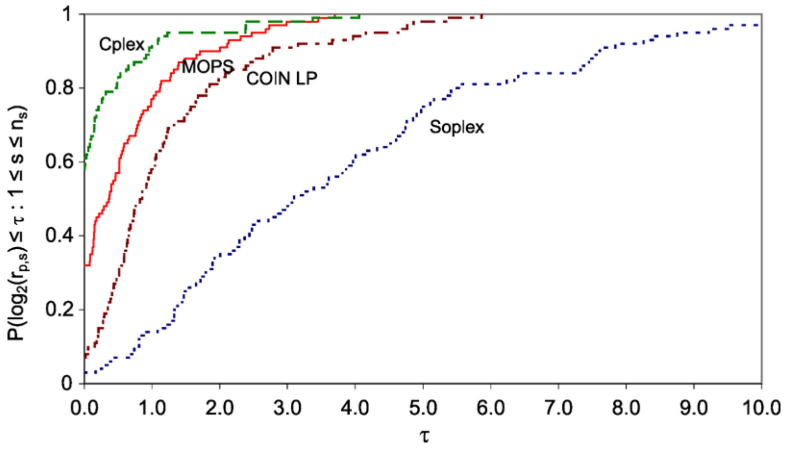

Fig. 1 Performance profile over solution time of 100 test models

wise computation of the transformed pivot row, if appropriate. Intermediate solution vectors such as $\boldsymbol{\rho}$, $\boldsymbol{\alpha}_q$ and $\boldsymbol{\alpha}^r$ were stored in dense arrays.

With this first version the model KEN-18 could be solved correctly, but our code was about 200 times slower in terms of solution time than other state-of-the-art solvers (like Cplex 8.0). The techniques presented above led to a particular breakthrough on the KEN-problems, which can now be solved in competitive time. The breakthrough in terms of numerical stability was achieved in version 7 with the incorporation of Harris' ratio test and cost shifting. This was the first version of our code which solved all of our test problems correctly.

To evaluate the performance of the MOPS dual simplex code compared to dual simplex implementations of other LP-systems (Soplex 1.2.1, COIN LP (CLP) 1.03.03 and Cplex 9.1) we solved the problems in our test set using the dual simplex method[8] of the four solvers with default settings and the respective system-specific LP preprocessing. The test runs were conducted under Windows XP Professional on a standard Intel Pentium IV PC with 3.2 GHz and 1 GB of main memory. The MOPS executable was built by the Intel Visual Fortran Compiler V9.0, the Soplex and CLP codes were compiled using Microsoft Visual C++ V6.0. Figure 1 shows the performance profiles over solution time. For a detailed description of the visualization technique of performance profiles we refer to [7]. Table 3 gives an overview of the solution times for five of the most difficult among our test set problems.

The performance profile shows that MOPS clearly outperforms CLP and Soplex, the latter of which fails on two of the largest problems (MUN1_M_D and MUN18_M_D).[9] On over 30% of the test models, MOPS achieves the best solution time and for about 80% of the problems it is at most two times slower than the best solver. However, on this benchmark, Cplex 9.1 clearly yields the best overall per-

[8] For Soplex, we used the entering algorithm.

[9] The 12 hour time limit is exceeded.

Table 3 Solution time (including time for LP preprocessing) on five difficult models.

	MOPS 7.9	Soplex 1.2.1	CLP 1.03.03	Cplex 9.1
MUN1_M_D	16265	>72000	12940	14295
MUN18_M_D	2733	>72000	3309	5469
RAIL4284	5766	10058	7417	4019
STP3D	2662	6726	1253	1021
STORMG2_1000	3448	15550	1747	537

formance. We want to emphasize again that neither the techniques used in the COIN LP code nor the internals of the Cplex code are documented in the research literature.

5 Conclusion

In this paper we presented the mathematical algorithms, computational techniques and implementation details, which are key performance factors for our dual simplex code. We described how to exploit hypersparsity in the dual simplex algorithm, gave a conceptual integration of Harris' ratio test, bound flipping and cost shifting techniques and described a sophisticated and efficient implementation. We also addressed important issues of the implementation of dual steepest edge pricing. Finally we showed on a large set of practical large scale LP problems, that our dual simplex code outperforms the best existing open-source and research codes and is competitive to the leading commercial LP-systems on our most difficult test problems.

Future research in this field can pursue several possible directions. One direction is to try to achieve further improvements in the given mathematical setting. Often inconspicuous implementation details have a great impact on the performance of the code (such as the bound handling after LP preprocessing). One important aspect is for instance to find better criteria for a problem specific switching between the different computational techniques (dense, sparse, hyper-sparse linear systems; simple and bound flipping ratio test etc.). Another possible research direction is to improve the dual simplex algorithm on the mathematical side. Pan proposed a new promising formulation of the dual simplex algorithm in [29]. It is however not clear yet, whether his algorithm is computationally competitive. Hager and Davis [6, 13] present a dual active set algorithm and show that it is competitive for a significant set of problems.

References

1. Benichou, M., Gautier, J., Hentges, G., Ribiere, G.: The efficient solution of large-scale linear programming problems. Math. Program. **13**, 280–322 (1977)
2. Bixby, R.: Solving real-world linear programs: a decade and more of progress. Oper. Res. **50**(2), 3–15 (2002)
3. Bixby, R.E., Martin, A.: Parallelizing the dual simplex method. INFORMS J. Comput. **12**(1), 45–56 (2000)
4. Bixby, R.E., Fenelon, M., Gu, Z., Rothberg, E., Wunderling, R.: MIP: Theory and practice closing the gap. In: Powell, M.J.D., Scholtes, S. (eds.) System Modelling and Optimization: Methods, Theory and Applications, pp. 19–49. Kluwer, Dordrecht (2000)

5. Bpmpd test problems. http://www.sztaki.hu/~meszaros/bpmpd/
6. Davis, T.A., Hager, W.W.: A sparse proximal implementation of the LP dual active set algorithm. Math. Program. **112**, 275–301 (2008). doi:10.1007/s10107-006-0017-0
7. Dolan, E.D., More, J.J.: Benchmarking optimization software with performance profiles. Math. Program. **91**(2), 201–213 (2002)
8. Dsor test problems. http://dsor.upb.de/koberstein/lptestset/
9. Forrest, J.J., Goldfarb, D.: Steepest-edge simplex algorithms for linear programming. Math. Program. **57**(3), 341–374 (1992)
10. Fourer, R.: Notes on the dual simplex method. Draft Report (1994)
11. Gilbert, J.R., Peierls, T.: Sparse partial pivoting in time proportional to arithmetic operations. SIAM J. Sci. Stat. Comput. **9**, 862–874 (1988)
12. Gill, P.E., Murray, W., Saunders, M.A., Wright, M.H.: A practical anti-cycling procedure for linearly constrained optimization. Math. Program. **45**, 437–474 (1989)
13. Hager, W.W.: The dual active set algorithm and its application to linear programming. Comput. Optim. Appl. **21**, 263–275 (2002)
14. Hall, J.A.J., McKinnon, K.I.M.: Hyper-sparsity in the revised simplex method and how to exploit it. Comput. Optim. Appl. **32**(3), 259–283 (2005)
15. Harris, P.M.J.: Pivot selection methods of the DEVEX LP code. Math. Program. **5**, 1–28 (1973)
16. Kirillova, F.M., Gabasov, R., Kostyukova, O.I.: A method of solving general linear programming problems. Doklady AN BSSR **23**(3), 197–200 (1979) (In Russian)
17. Koberstein, A.: The dual simplex method: Techniques for a fast and stable implementation. Technical Report, University of Paderborn, II—Fakultät für Wirtschaftswissenschaften/Department Wirtschaftsinformatik (2005). http://ubdata.uni-paderborn.de/ediss/05/2005/koberste/
18. Koberstein, A., Suhl, U.H.: Progress in the dual simplex algorithm for solving large scale LP problems: Practical dual phase 1 algorithms. Comput. Optim. Appl. **37**(1), 49–65 (2007)
19. Kostina, E.: The long step rule in the bounded-variable dual simplex method: Numerical experiments. Math. Methods Oper. Res. **55**, 413–429 (2002)
20. Lemke, C.E.: The dual method of solving the linear programming problem. Nav. Res. Logist. Q. **1**, 36–47 (1954)
21. Lougee-Heimer, R., et al.: The coin-or initiative: Open-source software accelerates operations research progress. ORMS Today **28**(5), 20–22 (2001). http://www.coin-or.org
22. Maros, I.: A piecewise linear dual procedure in mixed integer programming. In: Giannesi, et al. (eds.) New Trends in Mathematical Programming, pp. 159–170. Kluwer, Dordrecht (1998)
23. Maros, I.: A general phase-i method in linear programming. Eur. J. Oper. Res. **23**, 64–77 (1986)
24. Maros, I.: A generalized dual phase-2 simplex algorithm. Eur. J. Oper. Res. **149**(1), 1–16 (2003)
25. Maros, I.: Computational Techniques of the Simplex Method. Kluwer's International Series. Kluwer, Dordrecht (2003)
26. Mittelmann test problems. ftp://plato.asu.edu/pub/lpfree.html
27. Miplib 2003 test problems. http://miplib.zib.de/
28. Netlib test problems. http://www.netlib.org/lp/data/
29. Pan, P.Q.: A dual projective pivot algorithm for linear programming. Comput. Optim. Appl. **29**(3), 333–346 (2004)
30. Suhl, U.H.: Mops—mathematical optimization system. OR News **8**, 11–16 (2000). http://www.mops-optimizer.com
31. Suhl, U.H., Suhl, L.M.: Computing sparse lu factorizations for large-scale linear programming bases. ORSA J. Comput. **2**(4), 325–335 (1990)
32. Suhl, L.M., Suhl, U.H.: A fast lu-update for linear programming. Ann. Oper. Res. **43**, 33–47 (1993)
33. Wunderling, R.: Paralleler und objektorientierter simplex. Technical Report, Konrad-Zuse-Zentrum für Informationstechnik Berlin (1996)