Permutations in the factorization of simplex bases

Ricardo Fukasawa, Laurent Poirrier
{rfukasawa,lpoirrier}@uwaterloo.ca *

December 13, 2016

Abstract

The basis matrices corresponding to consecutive iterations of the simplex method only differ in a single column. This fact is commonly exploited in current LP solvers to avoid having to compute a new factorization of the basis at every iteration. Instead, a previous factorization is updated to reflect the modified column. Several methods are known for performing the update, most prominently the Forrest-Tomlin method. We present an alternative algorithm for the special case where the update can be performed purely by permuting rows and columns of the factors. In our experiments, this occurred for about half of the basis updates, and the new algorithm provides a modest reduction in computation time for the dual simplex method.

1 Introduction

Let a linear programming problem be given as

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & x \in \mathbb{R}^n_+, \end{aligned} \tag{1}$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. The simplex method finds a finite optimal solution x^* to (1) if such a solution exists. It exploits two central results from linear programming theory. First, the feasible region $P := \{x \in \mathbb{R}^n_+ : Ax = b\}$ is a polyhedron, and if the optimum is finite, then at least one vertex of P is an optimal solution. Secondly, every vertex of P is a basic feasible solution of (1), i.e. it can be obtained through a *basis* of (1). A basis $\mathcal{B}$ of (1) is a subset of size m of the column indices $\{1, \ldots, n\}$, such that the corresponding columns of A form an invertible matrix B. Note that the term basis is often used to designate both $\mathcal{B}$ and B.

We do not describe the simplex method here, and instead focus on a few aspects (sometimes called numerical *kernels*) of the linear algebra involved in the steps of the algorithm. We refer the interested reader to Chvátal [3]

*Partially supported by NSERC Discovery Grant RGPIN-05623 and MRI's Early Researcher Award ER11-08-174

for a comprehensive introduction to the simplex method, and to Maros [17] for a detailed description of its implementation details.

A central requirement of the simplex method is the ability to solve determined linear systems involving a linear programming basis. These systems are usually solved by computing an LU factorization of B, i.e. $LU = B$ where L is lower triangular and U is upper triangular.

At a given iteration $t+1$ of the simplex method, the current basis $B^{(t+1)}$ differs from the previous one $B^{(t)}$ in only one column. This fact can be exploited to modify the factorization $B^{(t)} = L^{(t)}U^{(t)}$ and obtain a factorization of $B^{(t+1)}$ that can be used to solve linear systems. The objective is to solve the linear systems involving $B^{(t+1)}$ at a lower computational cost than would be incurred with a fresh LU factorization of $B^{(t+1)}$.

There are multiple known methods to build such an updated factorization. As a first step, one can easily obtain a factorization of the new basis where one of the factors remains unchanged, while the other loses triangularity (see e.g. [4, 23]). The various update methods differ in the structure of the nontriangular factor and in how to restore its triangularity. We propose a method that finds a permutation of its rows and columns that is triangular, whenever such a permutation exists. As such, our approach is less general than existing ones: it fails when no triangular permutation exists, in which case we must fall back on one of the other methods. However, basis matrices are known to have a special structure where permutations of rows and columns can almost yield triangularity. We will show that, because of this, a triangular permutation does indeed often exist.

The outline of the paper is as follows. In Section 2, we describe this structure and we explain how it is currently exploited by linear programming solvers. Though this is part of the folklore in linear programming, to the best of our knowledge, the results presented in this section were never formalized. We discuss them in order to introduce some further concepts and for the sake of clarity. In Section 3, we present previous approaches to obtaining updated factorizations and then present our algorithm, whose particularity is that it takes advantage of the sparsity of the factors. As a consequence, it can have a very limited computational cost, given a careful implementation (described in Section 4). Finally, Section 5 presents our computational results. There, we confirm that (a) our method applies to many simplex iterations, (b) its computational cost is indeed small, and (c) it has a beneficial impact on the execution speed of the simplex method overall.

2 Factorizing with small nuclei

Since computing the factorization is an important step of the simplex method, problem structure is exploited to speed up its implementation. For instance, in the overwhelming majority of linear programming formulations encountered in practice, the matrix A is sparse. As an example, the average nonzero density of the 87 problems in the MIPLIB 2010 [15] benchmark set is 1.62% (all but 5 of the instances have a density below 5%, and those 5 instances all have fewer than 50000 nonzeros, while the overall average is above ten times more). Since basis matrices are formed of a subset of the columns of A, they are typically sparse as well.

Moreover, linear programming bases have one interesting property that separates them from the sparse matrices usually found in other scientific applications. Let us consider a *permutation* of the rows and columns of B that takes the form

$$P^T BQ = \left(\begin{array}{c|c|c} U & * & * \\ \hline 0 & L & 0 \\ \hline 0 & * & G \end{array}\right) \tag{2}$$

where U is upper triangular, L is lower triangular, and P and Q are permutation matrices. The parts of the matrix marked with a $*$ can have any structure. The remaining non-triangular part G is called the *nucleus*. We call this form *pseudo triangular*. In order to obtain an LU factorization of a matrix of the form (2), it is sufficient to factorize the nucleus, i.e. compute $L^G U^G = G$ by regular Gaussian elimination. Then,

$$P^T BQ = \left(\begin{array}{c|c|c} U & U' & U'' \\ \hline 0 & L & 0 \\ \hline 0 & L' & G \end{array}\right) = \left(\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & L & 0 \\ \hline 0 & L' & L^G \end{array}\right) \cdot \left(\begin{array}{c|c|c} U & U' & U'' \\ \hline 0 & I & 0 \\ \hline 0 & 0 & U^G \end{array}\right). \tag{3}$$

It is part of the folklore among simplex practitioners that one can typically find permutations P and Q such that the nucleus G is extremely small. This fact was known to Orchard-Hayes as early as 1968 [19], and Suhl and Suhl describe the implementation details of a procedure to exploit it [24]. The scientific literature on the subject is relatively scarce however, and there was little numerical data on the topic, until a recent computational survey by Luce et al. [16].

Luce et al. [16] quantify the numerical properties of the basis matrices occurring in the resolution of a large collection of LP instances. They use the Soplex code [26] both as a simplex solver (to sample simplex bases), and as a reference implementation for the factorization. As every modern simplex code, Soplex implements a sparse direct LU factorization with some variant of Markowitz pivoting [26]. The intent of Luce et al. [16] is to compare the traditional (in the context of linear programming) LU factorizer of Soplex with several state-of-the-art generic methods. Their results provide rigorous data that confirm the "folklore wisdom" in the field. Basis matrices are indeed typically sparse, and so are the L and U factors. Also, nuclei are small, especially for larger problems: for every single instance from their testset with more than 300000 constraints, the average nucleus size (over all bases factorized) was less than 4% of the basis size. They further show that the relative fill-in of factors obtained with Soplex is close to minimal. In other words, dynamic Markowitz pivoting generates factors that are almost as sparse as the sparsest possible factors (note that while even a dense factorization can be performed in $O(m^3)$ operations, finding the sparsest one is NP-hard [27]). As a consequence, for basis matrices, the traditional factorizer included in Soplex consistently outperforms even the most elaborated generic LU codes.

We now show how the permutations P and Q are found, and what guarantees they offer when constructed appropriately. All the results in this section are direct and well known among simplex practitioners, but we need to introduce some formalism in order to clarify the subsequent exposition.

Definition 1. A square matrix $H \in \mathbb{R}^{m\times m}$ is said to be in pseudo triangular form if

$$H = \left(\begin{array}{c|c|c} U & * & * \\ \hline 0 & L & 0 \\ \hline 0 & * & G \end{array}\right) \tag{4}$$

where U is a square upper triangular matrix, L is a square lower triangular matrix and G is a square matrix. We call G the *nucleus* of H.

Note that any matrix is immediately in pseudo triangular form, since we allow the U and L block submatrices to be empty, in which case $H = G$. However, as mentioned previously, we are interested in writing matrices in pseudo triangular form with nuclei of small sizes.

In this paper, we exclusively consider the factorization of invertible matrices, since basis matrices are always nonsingular. For such matrices, we can use the following property.

Lemma 1. *Let H be a pseudo triangular matrix partitioned as in* (4)*. Then,* $\det(H) = \det(U)\det(L)\det(G)$*. In particular, if H is nonsingular, then so are U, L and G. Moreover, this implies that all diagonal elements of U and L are nonzero.*

Proof. Observe the LU factorization of a pseudo triangular matrix given in (3). Both factors are block triangular. Their determinant is therefore the product of the determinants of the diagonal blocks, yielding $\det(H) = \det(I)\det(L)\det(L^G)\det(U)\det(I)\det(U^G)$ where $G = L^G U^G$. Since $\det(L^G)\det(U^G) = \det(G)$, the result follows. □

Definition 2. Let H be an $m \times m$ matrix. The element $H_{ij} \neq 0$ is called a *column-singleton* if $H_{lj} = 0$ for all $l \neq i$, i.e. if H_{ij} is the only nonzero element in its column. Column j of H is then said to be a *singleton-column*. Similarly, H_{ij} is a *row-singleton* if $H_{ik} = 0$ for all $k \neq j$, in which case row i is a *singleton-row*.

Finding a pseudo triangular permutation is straightforward. It simply consists in moving all column- and row-singletons to the front of the matrix. For instance, given a matrix H that has a column-singleton H_{ij} (Figure 1a), we can in slide column j is into the first position, moving columns $1, \ldots, j-1$ to the right by one place (Figure 1b), then slide row i into the first position, moving rows $1, \ldots, i-1$ down one place (Figure 1c). The resulting matrix H' has a column-singleton in the first position H'_{11}. It is thus pseudo triangular with an upper triangular part of size 1. The process can then be iterated by eliminating the first row and column and considering the remaining submatrix. Remark that while one column-singleton is eliminated at every iteration, all other column-singletons in H remain column-singletons in H'. Moreover, the elimination of one row from H may create new column-singletons in the remaining submatrix. Algorithm 1 formalizes this method.

Algorithm 1. Permutation of the rows and columns of a matrix into a pseudo triangular form.

Input: $B \in \mathbb{R}^{m\times m}$ nonsingular.
Initialize: Set $B^{(0)} := B$ and $k := 0$.

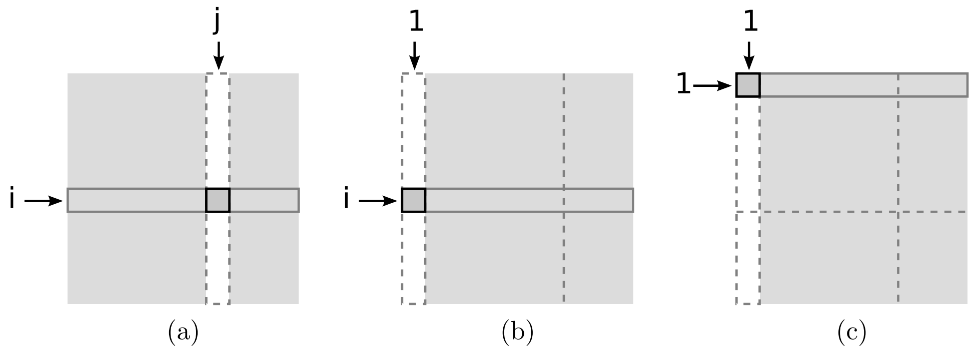


Figure 1: Sliding a column-singleton into U^p.

Step 1: Let $B^{(k)}$ be partitioned into

$$B^{(k)} = \left(\begin{array}{c|c} U^{(k)} & * \\ \hline 0 & G^{(k)} \end{array}\right),$$

where $U^{(k)} \in \mathbb{R}^{k\times k}$. If $G^{(k)}$ has no column-singleton, then set $\kappa := k$ and go to Step 2. Otherwise, let $G^{(k)}_{ij}$ be the only nonzero in column j of $G^{(k)}$, and set

$$B^{(k+1)} := \left(\begin{array}{c|c} U^{(k)} & * \\ \hline 0 & F \end{array}\right),$$

where $F = P^T G^{(k)} Q$, $P = (e_i|e_1|\dots|e_{i-1}|e_{i+1}|\dots|e_{m-k})$, and $Q = (e_j|e_1|\dots|e_{j-1}|e_{j+1}|\dots|e_{m-k})$. Clearly, $F_{i1} = 0$ for all $i \geq 2$, so the partition of $B^{(k+1)}$ at the next iteration of Step 1 will have the appropriate structure (i.e. zeros below $U^{(k+1)}$). Set $k := k+1$ and go to Step 1.
Step 2: Let $B^{(k)}$ be partitioned into

$$B^{(k)} = \left(\begin{array}{c|c|c} U^{(\kappa)} & * & * \\ \hline 0 & L^{(k)} & 0 \\ \hline 0 & * & G^{(k)} \end{array}\right),$$

where $L^{(k)} \in \mathbb{R}^{(k-\kappa)\times(k-\kappa)}$. If $G^{(k)}$ has no row-singleton, then set $\lambda := k$ and go to Step 3. Otherwise, let $G^{(k)}_{ij}$ be the only nonzero in row i of $G^{(k)}$, and set

$$B^{(k+1)} := \left(\begin{array}{c|c|c} U^{(\kappa)} & * & * \\ \hline 0 & L^{(k)} & 0 \\ \hline 0 & * & F \end{array}\right),$$

where $F = P^T G^{(k)} Q$, $P = (e_i|e_1|\dots|e_{i-1}|e_{i+1}|\dots|e_{m-k})$, and $Q = (e_j|e_1|\dots|e_{j-1}|e_{j+1}|\dots|e_{m-k})$. Again, $F_{1j} = 0$ for all $j \geq 2$, so the partition of $B^{(k+1)}$ at the next iteration of Step 2 will have the appropriate structure (i.e. zeros right of $L^{(k+1)}$). Set $k := k+1$ and go to Step 2.
Step 3: The result is $B^{(\lambda)}$, a pseudo triangular matrix with an upper triangular part of size $\kappa \times \kappa$, a lower triangular part of size $(\lambda-\kappa)\times(\lambda-\kappa)$, and a nucleus of size $(m-\lambda)\times(m-\lambda)$.

From a computational perspective, Step 1 of Algorithm 1 may be implemented as described in Pseudocode 1. The process is symmetric for Step 2, eliminating singleton-rows from the matrix. We finish this section by showing that Algorithm 1 indeed computes the smallest possible nucleus. To the best of our knowledge, we provide the first formal proof of this result, although the result itself is widely known and exploited.

```
For all j, compute nz[j], the number of nonzeros in column B_j.
Compute the set S := {j : nz[j] = 1} of column-singleton indices in B
k := 0
while S is not empty {
        Let j ∈ S, and i be such that B_ij is the nonzero of B_j
        S := S \ {j}
        row_bwd[k] := i
        col_bwd[k] := j
        k := k + 1
        for every nonzero B_il in row i of B {
                nz[l] := nz[l] − 1
                if nz[l] = 1 then S := S ∪ {l};
        }
}
Form U^(k) with rows row_bwd[1, . . . , k] and columns col_bwd[1, . . . , k] of B.
```

Pseudocode 1: Permuting B into a pseudo triangular form (Step 1).

Theorem 1. *Given a square nonsingular matrix B, Algorithm 1 yields a pseudo triangular permutation $B^{(\lambda)}$ of B with a nucleus of minimum size.*

Proof. We apply Algorithm 1 on B and obtain the matrix $B^{(\lambda)}$. For conciseness, we denote by $\mathcal{U}$, $\mathcal{L}$ and $\mathcal{G}$ the index subsets corresponding to the upper triangular, lower triangular, and nucleus parts of $B^{(\lambda)}$, respectively, i.e. $\mathcal{U} := \{1, \dots, \kappa\}$, $\mathcal{L} := \{\kappa+1, \dots, \lambda\}$ and $\mathcal{G} := \{\lambda+1, \dots, m\}$. Let B^* be a pseudo triangular permutation of B with a nucleus of minimum size. The sets $\mathcal{U}^*$, $\mathcal{L}^*$, and $\mathcal{G}^*$ are defined for B^* similarly to their counterparts for B. Note that, for $\mathcal{U}^*$ fixed, $\mathcal{L}^*$ is maximal i.e. there are no singleton-rows in the submatrix formed with rows and columns $\mathcal{G}^*$ of B^*. The contrary would immediately contradict the assumption that $\mathcal{G}^*$ is minimum. Furthermore, we may assume without loss of generality that $\mathcal{U}^*$ is maximal too, i.e. there are no singleton-columns in the submatrix formed with rows and columns $\mathcal{L}^* \cup \mathcal{G}^*$ of B^*. Indeed, any such singleton-column can be moved to $\mathcal{U}^*$ without affecting the size of $\mathcal{G}^*$. Algorithm 1 ensures that $\mathcal{U}$ and $\mathcal{L}$ are also maximal in the same sense. We then define the functions $r^*, c^* : \{1, \dots, m\} \to \{1, \dots, m\}$, such that the row $r^*(i)$ of B^* corresponds to the row i of B, and the column $c^*(j)$ of B^* corresponds to the column j of B. Recall that by Lemma 1, every diagonal element in the triangular blocks of $B^{(\lambda)}$ and B^* is nonzero. Thus, in the following, "diagonal" will always imply "nonzero". The proof works in three steps.

(i) We show that every column in $\mathcal{L} \cup \mathcal{G}$ maps to a column in $\mathcal{L}^* \cup \mathcal{G}^*$, i.e. $j \in \mathcal{L} \cup \mathcal{G}$ implies $c^*(j) \in \mathcal{L}^* \cup \mathcal{G}^*$. Let $j_a = \operatorname{argmin}_{j \in \mathcal{L} \cup \mathcal{G}} c^*(j)$. Suppose that the claim is not true, i.e. suppose that there exists $j \in \mathcal{L} \cup \mathcal{G}$ such that $c^*(j) \in \mathcal{U}^*$. Then, $c^*(j_a) \in \mathcal{U}^*$ (Figure 2). Since $j_a \in \mathcal{L} \cup \mathcal{G}$ and $\mathcal{U}$ is maximal, there are at least two nonzero elements in column j_a of $B^{(\lambda)}$. All of them correspond to elements of $\mathcal{U}^*$, so at least one lies above the diagonal of $\mathcal{U}^*$. Let that element be in row i_a of $B^{(\lambda)}$, corresponding to the element of B^* in row $r^*(i_a)$ and column $c^*(j_a)$, with $r^*(i_a) < c^*(j_a)$. The diagonal element in that row of $\mathcal{U}^*$ is in column $c^*(j_b) = r^*(i_a)$ for some j_b.

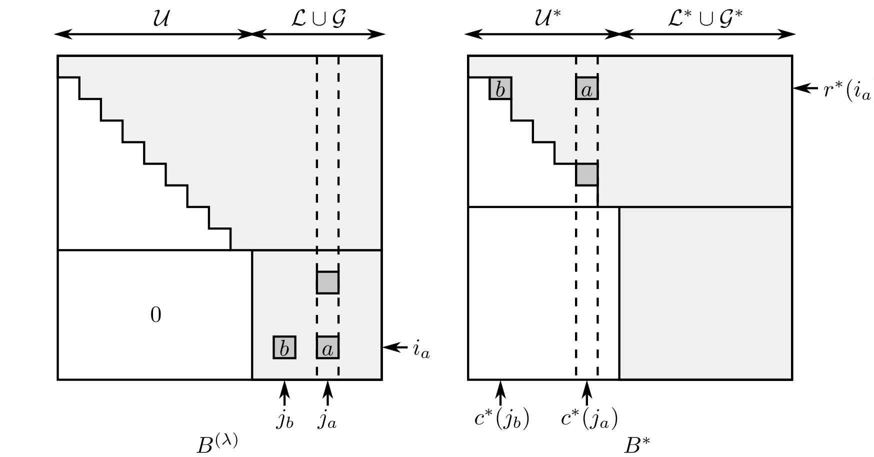


Figure 2: Proof of Theorem 1, step (i).

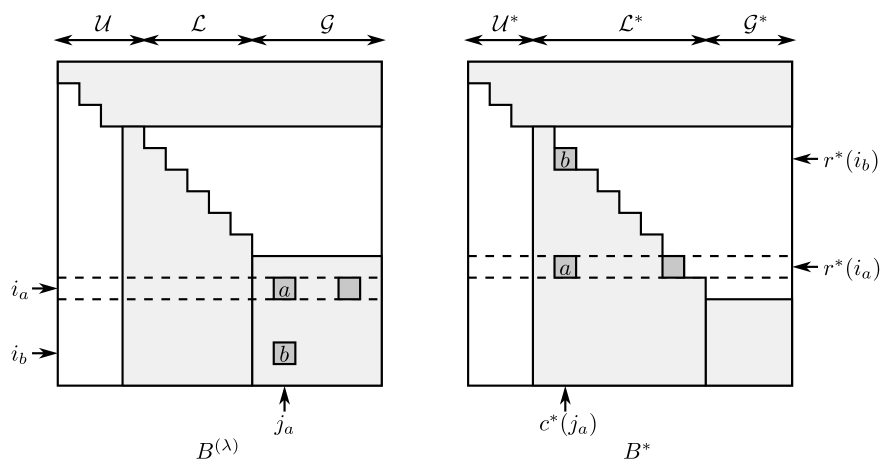


Figure 3: Proof of Theorem 1, step (iii).

Since it is a nonzero in the row i_a of $B^{(\lambda)}$, we know that $j_b \in \mathcal{L} \cup \mathcal{G}$. However, because $c^*(j_b) = r^*(i_a) < c^*(j_a)$, this contradicts the construction of j_a as $\operatorname{argmin}_{j \in \mathcal{L} \cup \mathcal{G}} c^*(j)$.

(ii) We can reverse the roles of $B^{(\lambda)}$ and B^* in the proof of (i). Therefore, every column in $\mathcal{L}^* \cup \mathcal{G}^*$ maps to a column in $\mathcal{L} \cup \mathcal{G}$, i.e. $c^*(j) \in \mathcal{L}^* \cup \mathcal{G}^*$ implies $j \in \mathcal{L} \cup \mathcal{G}$. Together, (i) and (ii) prove that $\mathcal{U}$ is a permutation of $\mathcal{U}^*$.

(iii) We transpose the reasoning that led to (i) and (ii) and apply it to the submatrices formed with the columns $\mathcal{L}^* \cup \mathcal{G}^*$ of B^* and $\mathcal{L} \cup \mathcal{G}$ of $B^{(\lambda)}$ (Figure 3). This yields $|\mathcal{L}| = |\mathcal{L}^*|$ and hence $|\mathcal{G}| = |\mathcal{G}^*|$, completing our proof. □

Corollary 1. *If there exists a permutation of the rows and columns of B that is upper triangular, then Algorithm 1 finds such a permutation $B^{(m)}$ when it reaches Step 2.*

Operation	Time (% of overall solution time, average over instances)
Simplex method	100%
- LU factorization	58.01%
- Permutation	25.20%
- Gaussian elimination	20.10%
- other	12.71%

Table 1: Refactorization at every iteration in our code.

Proof. By Theorem 1, Algorithm 1 yields a pseudo triangular matrix $B^{(m)}$ with no nucleus. Assume that $\kappa < m$ when the algorithm starts Step 2. Then it means that $G^{(\kappa)}$ does not contain a column-singleton. However, since the final result $B^{(m)}$ has no nucleus, there exists a permutation $L^{(m)}$ of $G^{(\kappa)}$ that is lower triangular, contradicting the absence of a column-singleton in $G^{(\kappa)}$. Hence $\kappa < m$ is impossible. □

3 Exploiting basis changes and updating a factorization

In this section we present our main contribution, which is a new method to exploit the particular structure of basis updates in the simplex algorithm. We start by presenting what was previously done and then present our idea.

3.1 Background and previous work

The method presented in the previous section is conceptually very simple, but it is applied to the whole basis matrix, while Gaussian elimination is only performed on the nucleus. As a result, it represents a sizable portion of the computational effort dedicated to the LU factorization, as shown on Table 1 (the complete data are presented in Table 6 and the conditions of the experiment are discussed in Section 5). When we force our code to compute a new factorization at each iteration, it spends 58.01% of the solution time computing factorizations, on average over the set of LP instances. This includes 25.20% of the overall solution time spent computing pseudo triangular permutations, despite the simplicity of Pseudocode 1, and only 20.10% of the overall time spent performing comparatively much more complex Gaussian eliminations.

On the other hand, from one iteration of the simplex method to the next, only one column of the basis matrix is modified. Several methods have been proposed to exploit this fact and *update* the factorization across consecutive iterations. These factorization updates are key to a successful implementation of the simplex method. This is emphasized in Table 2, which shows that forcing the CPLEX solver to recompute a new LU factorization at every iteration slows it down by a factor 3 to 4 (complete data in Table 5). The full LU factorization, despite being mostly performed via permutation, is not cheap enough to be computed at every iteration of the simplex method. This is somewhat counter-intuitive given the simplicity of the permuting process. But once an effective update method is implemented, the factorization becomes so much faster that it is not the bottleneck anymore. Then, more time is typically spent in the triangular solves.

Factorization update method	Iterations (geometric mean)	Time (s) (geometric mean, shift = 10s)	Time per iteration (ms) (geometric mean, shift = 10ms)
none (refactor)	1165.832	9.598	2.749
automatic	1220.191	2.894	0.595

Table 2: The impact of factorization update on CPLEX 12.6

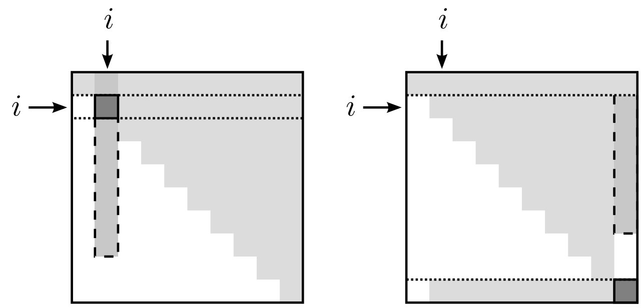


Figure 4: Pivoting W_{ii} in the Forrest-Tomlin update.

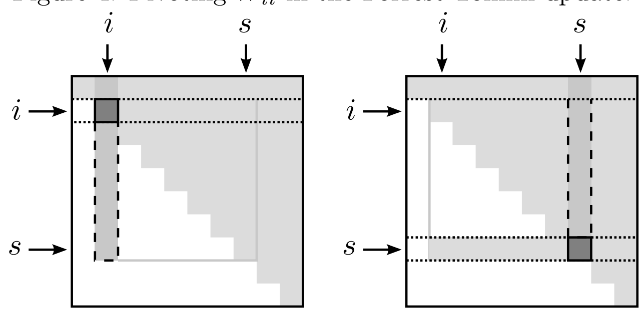


Figure 5: Pivoting W_{ii} in the Suhl-Suhl update.

The first update method for the LU factorization was proposed by Bartels and Golub in 1969 [1]. Subsequent alternatives were proposed by Forrest and Tomlin in 1972 [4], Saunders in 1976 [21, 22], Reid in 1982 [20], Suhl and Suhl in 1993 [23], and Huangfu and Hall in 2014 [11]. Update methods are also available for other representations of the basis inverse (e.g. for the product-form inverse [11]). As a result of his computational experience with CPLEX [25] and Gurobi [8], Bixby recommended the use of "some variant" of the Forrest-Tomlin update in 2009 [2]. We thus only focus on the Forrest-Tomlin update here, and its Suhl-Suhl refinement (the former can be seen as a simplified version of the latter). A good overview of all the different methods is provided by Chvátal [3], and recent updates on the implementation details of the Forrest-Tomlin and Suhl-Suhl updates are provided by Hall [9], Maros [17], Koberstein [13, 14] and Huangfu [10].

The Forrest-Tomlin approach starts with the following observation. Let B be the basis matrix at a given iteration. Assume that we have a factorization $B = LU$ of that matrix. At the next iteration, the basis matrix

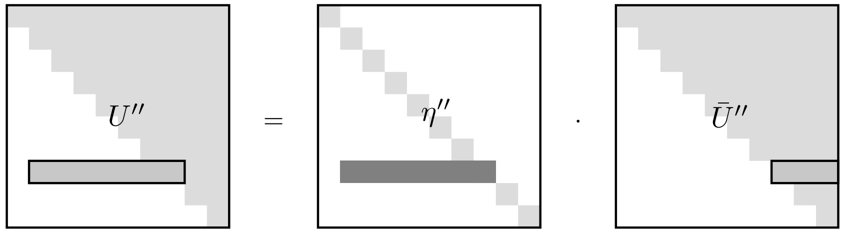


Figure 6: Factorization $W' = \eta'\bar{U}'$ via Gaussian elimination.

B^1 is the same as B except in column i, which is replaced by the entering column vector a_j. We can write

$$\begin{aligned} B^1 &= B - Be_ie_i^T + a_je_i^T \\ &= LU - LUe_ie_i^T + a_je_i^T \\ &= L\left(U - Ue_ie_i^T + L^{-1}a_je_i^T\right) \\ &= L\ W \end{aligned}$$

where $W = U - Ue_ie_i^T + L^{-1}a_je_i^T$ is, by construction, upper triangular except in column i. The ith column of W is called the *spike*, and W is thus called a *spiked* upper triangular matrix. The first operation we perform on W is to pivot the diagonal element of the spike to the back of the matrix (Figure 4). It is here that the Suhl-Suhl update refines Forrest-Tomlin, by only considering the diagonal submatrix that fully contains the spike as its leftmost column (Figure 5). We obtain a matrix $W' = P^TWQ$ that is upper triangular with a row spike (if P and Q are the appropriate permutation matrices). The row spike is then eliminated by Gaussian elimination, providing the factorization $W' = \eta'\bar{U}'$ (Figure 6) where $\bar{U}'$ is upper triangular and η' is diagonal except in one row (matrices of that form are commonly referred to as "row eta matrices"). We now consider the matrices $\eta := P\eta'$ and $\bar{U} := \bar{U}'Q^T$. It is easy to see that $W = \eta\bar{U}$. A factorization of B^1 is thus given by $B^1 = L\eta\bar{U}$.

The matrix $\bar{U}$ is not upper triangular, but we know that it can be permuted into a triangular matrix, and we keep track of the corresponding permutation matrices P and Q. That is enough to permit forward or backward substitution (FTRAN or BTRAN) and solve linear systems. Similarly, it is easy to solve with a permuted row eta matrix such as η. As will be clear below, performing the pivot is necessary in order to obtain a row spike, which can be eliminated through *premultiplication* by an eta matrix (postmultiplication would not allow us to iterate on the Forrest-Tomlin formula).

The process can be generalized to multiple consecutive iterations. Let $H^k := \eta^1 \cdots \eta^k$ and assume that $B^k = LH^kU^k$. The first basis in the sequence is obtained by computing a factorization of $B^0 = LU^0$ as described in the previous section. We keep track of the permutation matrices P^k and Q^k that are such that ${P^k}^TU^kQ^k$ is upper triangular. We start with $P^0 := I$, $Q^0 := I$ and $H^0 := I$. We obtain the relations

$$\begin{aligned}
B^{k+1} &= B^k - B^k e_i e_i^T + a_j e_i^T \\
&= LH^kU^k - LH^kU^k e_i e_i^T + a_j e_i^T \\
&= LH^k \left(U^k - U^k e_i e_i^T + {H^k}^{-1} L^{-1} a_j e_i^T\right) \\
&= LH^k\, W^k \\
&= LH^k \eta^{k+1} U^{k+1} \\
&= LH^{k+1} U^{k+1}
\end{aligned}$$

where $\eta^{k+1}U^{k+1}$ is a factorization of $W^k = U^k - U^k e_i e_i^T + {H^k}^{-1} L^{-1} a_j e_i^T$. Specifically, we start with the column-spiked matrix ${P^k}^T W^k Q^k$ and pivot the diagonal element of the spike to the back. We obtain the row-spiked ${P^{k+1}}^T W^k Q^{k+1}$, on which we perform Gaussian elimination to obtain ${P^{k+1}}^T W^k Q^{k+1} = \eta' \bar{U}'$. We finally let $\eta^{k+1} := P^{k+1}\eta'$ and $U^{k+1} := \bar{U}' {Q^{k+1}}^T$, and verify that $\eta^{k+1}U^{k+1} = W^k$.

The different update methods let us avoid the computation of a fresh LU factorization at each iteration of the simplex method. We instead perform algebraic operations that are much less expensive computationally. The drawback is the accumulation of eta matrices in the factorization of the basis. This increases the time required to solve linear systems and decreases the numerical accuracy of the solutions to those systems. To compensate for this, simplex codes regularly perform fresh refactorizations of the basis matrix. That way, the number of eta matrices stays limited.

3.2 Exploiting sparsity

The objective of this paper is to detect when the spiked matrix W^k is actually upper triangular already, up to a permutation of its rows and columns. In such a case, the Forrest-Tomlin procedure can be skipped altogether for the current iteration, and no additional eta matrix has to be included in the factorization.

We described a method for detecting such occurrence in the previous section: we could simply slide all column-singletons to the front of the matrix, yielding a triangular permutation of W^k whenever one exists. This idea is not novel. Reid [20] proposed it as a first step to its method. We implemented this approach, but as shown in Table 3, it is not a practical one (details on Table 6). This result is to be expected in light of the previous experiments: A code computing an LU factorization at every iteration spends about 25.20% of its time just for finding a pseudo triangular permutation of basis matrices (Table 1), and only 20.10% performing Gaussian elimination. On the other hand, employing an LU update method makes the solver around 3 to 4 times faster overall (Table 2). It is thus natural that looking for a triangular permutation at every iteration using the same method is prohibitively slow. However, one surprising result arises from this experiment: On average, in our test, 53.897% of the W^k matrices could be permuted into a triangular matrix.

As opposed to a renewed application of Pseudocode 1, we propose an approach that exploits the sparsity of the basis matrices. To introduce the approach, let us first introduce a few definitions and provide some theoretical

Factorization update method	Iterations (geometric mean)	Time (s) (geometric mean, shift = 10s)	Time per iteration (ms) (geometric mean, shift = 10ms)
none (refactor)	1213.276	11.853	4.003
Suhl-Suhl	1229.210	4.014	0.797
Reid + Suhl-Suhl	1202.992	4.880	1.119

Table 3: The impact of factorization update in our code.

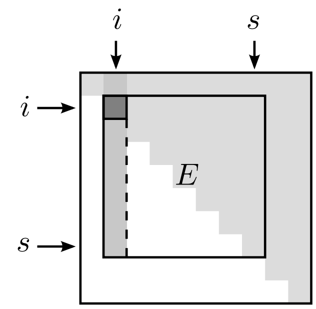


Figure 7: The submatrix E of W^k

results.

To start, note that since B is a basis, it is not singular. Neither are its factors in any given factorization, because $\det(B)$ is equal to the product of the determinant of its factors. Let $E \in \mathbb{R}^{s \times s}$ be the square diagonal submatrix of W^k such that its leftmost column is the spike (Figure 7). Despite not being triangular, W^k can be written in the block-triangular form

$$W^k = \left(\begin{array}{c|c|c} U^{11} & U^{12} & U^{13} \\ \hline 0 & E & U^{23} \\ \hline 0 & 0 & U^{33} \end{array}\right), \tag{5}$$

where U^{11} and U^{33} are upper triangular. It is then easy to see that E is an N-matrix, as defined in Definition 3.

Definition 3. A nonsingular matrix E is an N-matrix if $E_{ij} = 0$ for all $1 < j < i$ and $E_{ii} \neq 0$ for all $1 < i$.

Lemma 2 then shows that E is invertible, and that a triangular permutation of W^k exists if and only if one exists for E.

Lemma 2. *Let $H \in \mathbb{R}^{m \times m}$ be a nonsingular block upper triangular matrix of the form*

$$\left(\begin{array}{c|c|c|c} H^{11} & H^{12} & \cdots & H^{1\nu} \\ \hline 0 & H^{22} & \cdots & H^{2\nu} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline 0 & 0 & \cdots & H^{\nu\nu} \end{array}\right) \tag{6}$$

where H^{tt} are square matrices for all $t \in \{1, \ldots, \nu\}$. Then, H can be permuted into an upper triangular matrix

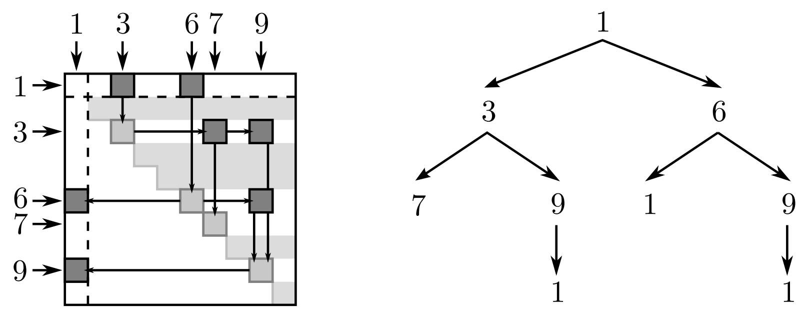


Figure 8: An N-matrix and its N-graph.

if and only if H^{tt} can be permuted into an upper triangular matrix for all $t \in \{1, \dots, \nu\}$.

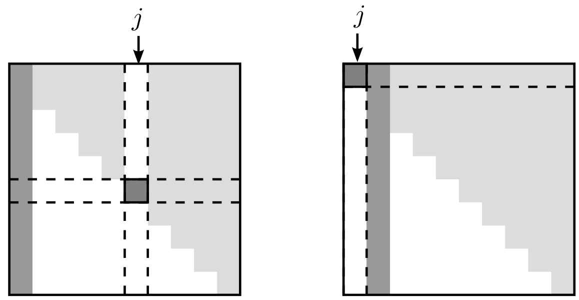
Proof. If: direct. Only if: Let H^* be a permutation of the rows and columns of H that is upper triangular. There exist permutation matrices P and Q such that $H = P^T H^* Q$. Observe that $|\det(H)| = |\det(P)\det(H^*)\det(Q)| = |\det(H^*)|$. Furthermore, $\det(H) = \prod_{t=1}^{\nu} \det(H^{tt})$ and $\det(H^*) = \prod_{k=1}^{m} H^*_{kk}$. The proof proceeds in two steps.

(i) We show that every diagonal element H^*_{kk} of H^* corresponds to an element in a diagonal block H^{tt} of H for some t. This is a property of the permutation matrices P and Q, and it can be proven as follows. We construct the matrix $H^\mu := H^* \circ \mu e_k e_k^T$, where $\circ$ denotes the Schur product and e_k is the kth column of the $m \times m$ identity matrix. Since H^μ is triangular, $\det(H^\mu) = \prod_{k=1}^{m} H^\mu_{kk} = \mu \det(H^*)$ for any value of $\mu \in \mathbb{R}$. The matrix H^μ also has zeros wherever H^* has, so $P^T H^\mu Q$ has the same block triangular structure as H. Suppose that H^*_{kk} does not correspond to an element in a diagonal block of H. Then $|\det(H^\mu)| = |\det(P^T H^\mu Q)| = |\prod_{t=1}^{\nu} \det(H^{tt})| = |\det(H)|$. Therefore, $|\mu \det(H)| = |\det(H)|$ for all $\mu \in \mathbb{R}$. This implies $\det(H) = 0$, which is a contradiction.

(ii) For any $\tau \in \{1, \dots, \nu\}$, we construct a triangular matrix $H^{*\tau\tau}$ that is a permutation of the rows and columns of $H^{\tau\tau}$. Observe that one can remove row k and column k from a triangular matrix and obtain a new triangular matrix. By (i), we know that every diagonal element H^*_{kk} of H^* corresponds to an element of H^{tt} for some t. For every k such that $t \neq \tau$, we remove row k and column k from H^*. The resulting matrix is the desired matrix $H^{*\tau\tau}$. Indeed, we removed from H^* exactly all the rows and columns that do not correspond to rows and columns of $H^{\tau\tau}$. It is thus a permutation of the rows and columns of $H^{\tau\tau}$. Furthermore, as mentioned earlier, $H^{*\tau\tau}$ is triangular by construction. □

We now focus on finding an upper triangular permutation of E and start by defining the N-graph on the nonzeros of E. An example of an N-matrix and its N-graph is given on Figure 8. Note that distinct nodes of the N-graph may share a same label.

Definition 4. Let E be an N-matrix. The *N-graph* of E is a directed tree constructed in the following way: The root node is a special element labeled 1. If a node labeled 1 is not the root node, then it is a leaf node, and it is called a *spike* leaf node. Otherwise, a node labeled i has one child for every nondiagonal nonzero element



(a) (b)

Figure 9: E_{jj} is a column-singleton that is pivoted to the front.

in row i of E, and each child is labeled with the corresponding column index, i.e. the children correspond to $\{j : E_{ij} \neq 0, i \neq j\}$.

Theorem 2 shows that a simple depth-first search on the N-graph of E is enough to find a triangular permutation of E whenever one exists. The proof is constructive and Pseudocode 2 shows how to build the permutation of the rows and columns. We claim that the method is superior to Pseudocode 1 because the N-graph is a sparse structure. It lets us consider only the row-singletons that we need to pivot in order to find a triangular permutation, as opposed to iteratively pivoting out all of them.

Theorem 2. *Let $E \in \mathbb{R}^{s\times s}$ be an N-matrix that is not in upper triangular form. Then E admits an upper triangular permutation if and only if its N-graph has exactly one spike leaf node.*

We first show in Lemma 3 that we may ignore rows and columns of E that are not covered by any label of its N-graph. Then, Lemma 4 shows that we can also ignore subtrees of the N-graph that do not contain spike leaf nodes, and all the rows and columns corresponding to the associated labels.

Lemma 3. *If there is no node labeled i in the N-graph of E, then row i and column i can be pivoted out of E. More precisely, there exists a permutation of the rows and columns of W^k yielding a partition of the form* (5) *where $E \in \mathbb{R}^{s\times s}$ and every index $i \in \{1, \dots, s\}$ appears in the labels of its N-graph.*

Proof. Let $\mathcal{I}$ be the set of all labels among $\{1, \dots, s\}$ that do not appear in the N-graph of E. We find the minimum element j of $\mathcal{I}$, i.e. $j := \min\{i \in \mathcal{I}\}$. Note that since the root node is labeled 1, $1 \notin \mathcal{I}$ so $j > 1$. Suppose that there exists a nonzero element E_{ij} in column j with $i \neq j$. Because $j > 1$ and E is an N-matrix, we know that $i < j$. Thus $i \notin \mathcal{I}$, so there is a node labeled i in the N-graph of E. But given that $E_{ij} \neq 0$ and $i \neq j$, by Definition 4, that node has a child labeled j, contradicting $j \in \mathcal{I}$. Therefore, E_{jj} is a column-singleton and we can pivot it to the front of E (Figure 9). We then set $\mathcal{I} := \mathcal{I} \setminus \{j\}$ and proceed until $\mathcal{I}$ is empty. □

Lemma 4. *Let T be a subtree of the N-graph of E rooted at a node labeled $r > 1$, consisting of r and all its descendants. If T has no spike leaf node, then row r and column r can be pivoted out of E. In other words, there exists a permutation of the rows and columns of W^k yielding a partition of the form* (5) *where all the leaf nodes in the N-graph of E are spike leaf nodes.*

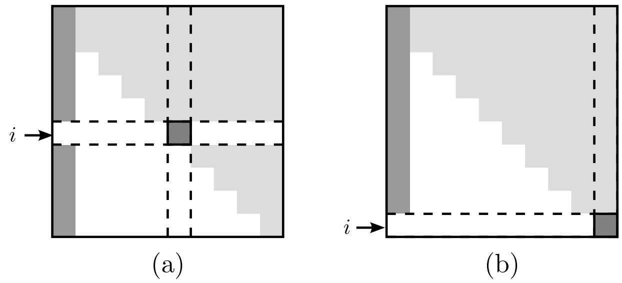


Figure 10: E_{ii} is a row-singleton that is pivoted to the back.

Proof. Pick a leaf node of T. Since it is not a spike leaf node, it has a label $i > 1$. The element E_{ii} is a row-singleton, and we can pivot it to the back of E (Figure 10), eliminating row i and column i. We then update E and its N-graph. Note that this does not create any new spike leaf nodes, so T will still have no spike leaf node. We may proceed until all rows and columns corresponding to labels of T are permuted out. Repeated application of the latter procedure for every non-spike leaf node proves the lemma. □

Corollary 2. *The N-graph of an N-matrix has at least one spike leaf node.*

Proof. Let E be an N-matrix. If there is no spike leaf node, then $E_{11} = 0$ and by Lemma 4, all the nondiagonal nonzero elements of the first row of E can be pivoted out. This contradicts the assumption that E is nonsingular. □

Proof of Theorem 2. The proof is constructive. First, we use Lemma 4 to permute out every subtree of the N-graph that does not contain a spike leaf node. We may now assume that every leaf node of the N-graph of E is a spike leaf node. If there is exactly one spike leaf node, then the N-graph is a path, otherwise it is a tree.

(i) We first assume that the root node has exactly one child. This is the case e.g. if the N-graph is a path. Let that child be labeled i. The element $E_{1i} \neq 0$ is a row-singleton in the first row. If the child is a spike leaf node, then $i = 1$ and we can slide E_{11} into the bottom-right position, directly obtaining a triangular matrix (Figure 11). Otherwise, we slide E_{1i} into the bottom-right position, then slide the spike column into position $i - 1$ (Figure 12). The resulting permuted matrix takes the form

$$P^T E Q = \left(\begin{array}{c|c|c} V^{(1)} & * & * \\ \hline 0 & E^{(1)} & \begin{array}{c} E_{ii} \\ \hline 0 \end{array} \\ \hline 0^T & 0^T & E_{1i} \end{array}\right)$$

where P and Q are permutation matrices, and $V^{(1)}$ is upper triangular. By construction, $E^{(1)}$ is an N-matrix of dimension $(s - i + 1) \times (s - i + 1)$. Furthermore, its first row corresponds to the ith row of E, minus the diagonal element E_{ii}. Therefore, the N-graph of $E^{(1)}$ is the subtree of the N-graph of E rooted at our initial

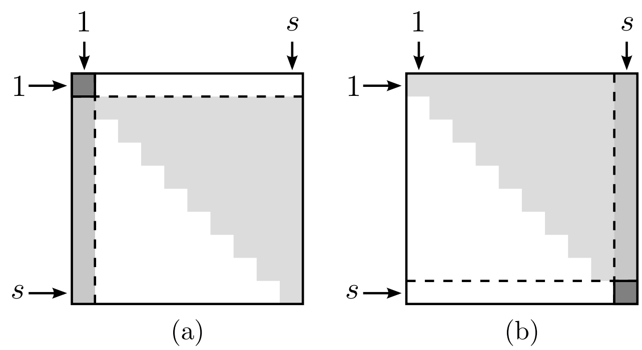


Figure 11: E_{11} is a row-singleton.

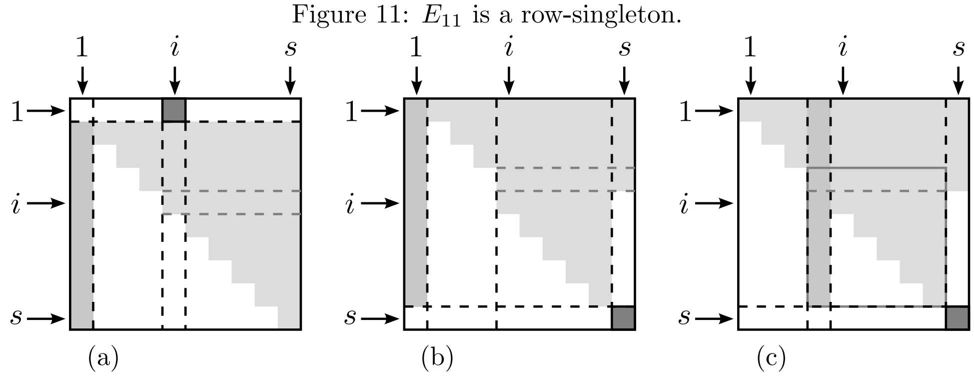


Figure 12: E_{1i} is a row-singleton.

child node labeled i (with nodes relabeled to follow the new indexing). By Lemma 2, there is a triangular permutation for E if and only if there is one for $E^{(1)}$. Proceeding with $E^{(1)}$, we obtain a finite sequence of matrices $E = E^{(0)}, E^{(1)}, E^{(2)}, \ldots, E^{(\tau)}$ of strictly decreasing size. If the N-graph of E is a path, then the root of the N-graph of $E^{(\tau)}$ has one child labeled 1, and we obtain a triangular permutation of E.

(ii) Otherwise, if the N-graph of E is a tree, then the root node of $E^{(\tau)}$ has two children, for some $\tau \geq 0$. We then use Lemma 3 to permute out every row and column i of $E^{(\tau)}$ that is absent from the labels of the N-graph of $E^{(\tau)}$, obtaining

$$P'^T E Q' = \left(\begin{array}{c|c|c} V' & * & * \\ \hline 0 & E' & * \\ \hline 0 & 0 & U' \end{array}\right)$$

where P' and Q' are permutation matrices, V' and U' are upper triangular, and $E' \in \mathbb{R}^{s' \times s'}$ has the same N-graph structure as $E^{(\tau)}$. By our use of Lemma 3, all the rows $\{2, \ldots, s'\}$ of E' have an associated label in its N-graph, and because we initially applied Lemma 4, all leaf nodes are spike leaf nodes. Therefore, E' has no singleton-row, so by Lemma 2, there is no triangular permutation of E. $\square$

The discussion in the proof of Theorem 2 directly yields an algorithm for finding the triangular permutation of E if such a permutation exists. An example implementation is described by Pseudocode 2. Calling `subtree`(E, 1) is equivalent to performing a depth-first search on the N-graph of E. It returns the number of spike leaf nodes and two lists of pivots P^0 and P^1. The pivots in P^0 correspond to the subtrees containing no spike leaf nodes, while the pivots in P^1 correspond to the path from the root to the spike leaf node, if unique. If there is exactly one spike leaf node, pivoting the elements in $P^0 \cup P^1$ to the back of the matrix performs the appropriate permutation. Note that the pivot lists are ordered, so we use the operator $\cup$ to designate an ordered concatenation that discards duplicate pivots from its right-hand side.

Pseudocode 2 exploits the sparsity of E by ignoring all the rows that have no label in its N-graph. In this sense, it does not use Lemma 3, which we only use as a theoretical tool to prove Theorem 2.

```
function (spike, P^1, P^0) = subtree(E, i)
{
        spike = 0
        P^0 = ∅
        P^1 = ∅
        for j : E_ij ≠ 0 {
                if j = 1 {
                        spike = spike + 1
                        P^1 = {(i, j)}
                } else if j > i {
                        (sub, S^1, S^0) = subtree(E, j)
                        P^0 = P^0 ∪ S^0
                        if sub ≥ 1 {
                                P^1 = {(i, j)} ∪ S^1
                                spike = spike + sub
                        }
                }
        }
        if spike = 0 {
                P^0 = P^0 ∪ {(i, i)}
        }
        return((spike, P^1, P^0))
}
```

Pseudocode 2: Finding a triangular permutation of an N-matrix E.

4 Implementation issues

Our code follows directly from the previous exposition. As Pseudocode 2 describes, we perform a depth-first search on the N-graph of the N-matrix E. The graph is not explicitly stored in memory but instead arises implicitly from a packed sparse representation of the rows of E. We deviate from a straightforward implementation in that row i is maked as “explored” just before `subtree`(E, i) returns. This way, a row (and the corresponding subtree) is never traversed twice.

In practice, the simplest way to store the permutation of the U factor that yields a triangular matrix U' is to use four arrays of integers: If row i of U corresponds to row k of U', then `row_fwd`[i] = k and `row_bwd`[k] = i. Similarly, if column j of U corresponds to column k of U', then `col_fwd`[j] = k and `col_bwd`[k] = j. The output of our implementation of Pseudocode 2 is two ordered lists of pivots. The pivots can then be

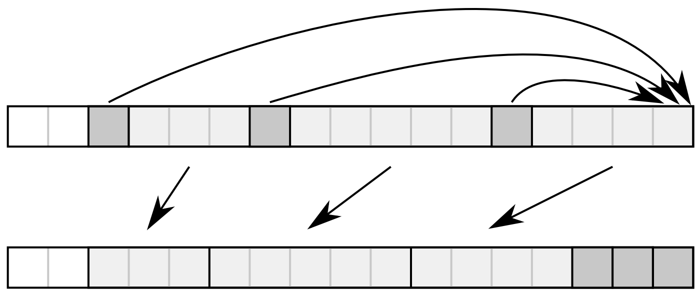


Figure 13: Performing $p = 3$ pivot on `row_bwd` is $O(n + p)$.

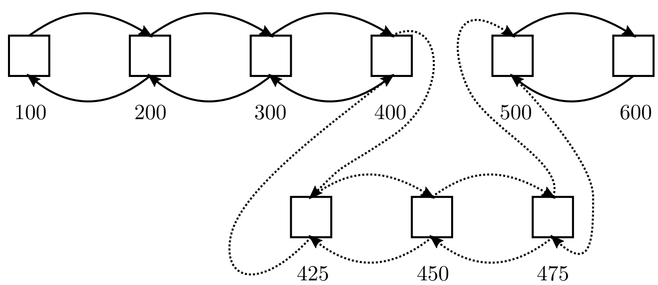


Figure 14: Inserting and labeling $p = 3$ elements in a linked list.

performed on `row_bwd` and `col_bwd`, while `row_fwd` and `col_fwd` are updated accordingly to reflect the change. As illustrated on Figure 13, p pivots can be performed simultaneously in $O(n + p)$ operations. However, this is a dense operation, and despite its apparent simplicity, it can be computationally expensive (especially for updating the `row_fwd` and `col_fwd` arrays, whose access pattern is not cache-friendly during the operation). Moreover, with the Forrest-Tomlin or Suhl-Suhl updates, only symmetric permutations are performed. The corresponding vectors `row_*` and `col_*` would always contain the same data, so in practice only one of them is used. As a consequence, whatever time is spent updating the permutation vectors in the Suhl-Suhl update, double that time would be spent with our method.

The computational cost of updating these dense row (and/or column) mappings is a known problem. Koberstein [14] mentions that the Coin-Clp code [5] features a method for doing it in constant time for the Forrest-Tomlin update, but it also states that whether an analogous method exists for the Suhl-Suhl update is an open question.

To mitigate this issue, we introduce a datastructure that represents `*_bwd` as a doubly-linked list. That way, deletions and insertions can be performed in constant time. One operation that is not $O(1)$ with linked lists is order comparison: given two list elements A and B, tell whether A is before B or B is before A in the list. This operation is necessary to implement sparse triangular solves. However, we can compensate for this shortcoming by adding a numeric label to each element. Whenever an element is inserted, it gets assigned a label that is strictly greater than that of the previous element, and strictly smaller than that of the next. When no such label exists (because of our representation of numbers with a finite number of bits), we need to relabel the whole list. Specifically, when inserting p elements, we divide the label range between the element preceding them and the one following them into $p+1$ intervals (Figure 14). In the worst case, we insert elements at the same place at every iteration of the simplex method, exploiting every time $\log(p)$ more bits in the representation of the labels. Assuming that we use 64-bit integers, that means relabeling every $(64-\log(n))/\log(p)$ iterations, where $\log(p)$ and $\log(n)$ are much smaller than 64, even for the largest instances. This is in the worst case though, and in practice, we will show that performing the pivots on this datastructure takes a very small fraction of the overall solution time.

A final refinement is that we store `col_*` and `row_*` information together in a single array of structures, described in Pseudocode 3. This should not improve performance in any significant way, but it simplifies some bookkeeping. Each `struct element` corresponds to a diagonal element; its members `i` and `j` indicate where to find it in U, the equivalent of `row_bwd[]` and `col_bwd[]` previously. Rows and columns can be enumerated in the order of U' (i.e. in the triangular order) by following the linked list, starting from `head` or `tail`. Rows (resp. columns) of U can be associated to elements of the linked list by using `row_fwd` (resp. `col_fwd`). Then, they can be compared to each other using their `label` member. Note that the exact indices of rows and columns of U' can not be obtained or used directly, but `label`, `prev` and `next` together fill that role.

```
struct permutation {
      struct element *head, *tail;

      struct element **row_fwd;
      struct element **col_fwd;
};
```

```
struct element {
      int i, j;
      unsigned long label;
      struct element *prev, *next;
};
```

Pseudocode 3: Datastructures for the representation of permutations.

This representation could be simplified if we adopt a small restriction consisting in always performing insertions at the end of the matrix, akin to what is done in the Forrest-Tomlin update. Then the `label` member of newly inserted elements could just be incremented from n. This would yield truly $O(1)$ insertions, at the cost of considering larger E matrices and performing more pivots. Having small integer values for `label` has further advantages, in particular in the implementation of the sparse triangular solves, but their discussion is beyond the scope of this paper.

5 Results

We perform our computational experiments on two sets of instances. First, we consider the root node LPs of problems in the MIPLIB 2010 [15] benchmark set. Because they are relevant in the context of the branch-and-bound method for mixed-integer programming, we first preprocess every instance using the MIP presolver of CPLEX 12.6 [25], then drop all integrality constraints. The second set is composed of various LP instances gathered by Mittelmann [18] for benchmarking purposes. The Mittelmann instances are harder to solve, so we use them only for confirming our final results on bigger instances (their solution times are typically a couple of orders of magnitude larger than the solution times for preprocessed MIPLIB 2010 instances).

We run our own implementation of the dual simplex method, with steepest-edge pricing ("dual algorithm 1" [6]) and a variant of the bound-flipping ratio test [12, 7]. No LP preprocessing was applied to the instances. Where possible, we also run the LP solver of CPLEX 12.6 to ensure that our conclusions are not particular to our code (Table 5). There, we use the dual simplex implementation of CPLEX with LP preprocessing disabled; all other parameters are kept to their defaults. All tests are performed on a computer with an Intel Core i5-3210M processor clocked at 2.50 GHz clock and 8 Gb of RAM. All running times shown for instances in the MIPLIB 2010 [15] benchmark set are averaged over three runs.

Across all tables, the label *it* denotes the number of iterations, *time* the total solution time in seconds, and *t/it* the time per iteration, in milliseconds. For *time* and *t/it*, the geometric means presented are *shifted* by a constant s, i.e.

$$\text{geom.mean}(t_1, \ldots, t_n) := \left(\prod_{i=1}^{n} (t_i + s) \right)^{1/n} - s.$$

The value of s is 10 in both cases: 10 seconds for *time* and 10 milliseconds for *t/it*.

Usefulness of factorization update methods. In Section 3, we discussed the computational cost of existing factorization update methods, with the support of aggregate experimental results in Tables 1, 2 and 3. We now present the details of these experiments, in Tables 5 and 6. They include only MIPLIB 2010 instances. Two variants of the CPLEX code are compared on Table 5: *CPLEX (refactor)* gives results for CPLEX with the maximum interval between refactorizations fixed to 1, effectively disabling the Forrest-Tomlin update, while *CPLEX (update)* corresponds to CPLEX in its default configuration. The main observation is that enabling the factorization update yields a dramatic drop in time per iteration, from 2.749ms to 0.595ms in shifted geometric mean. Table 6 shows similar outcomes with three variants of our code: in the first, *refactor*, there is no factorization update; the second, *Suhl-Suhl update*, is the default configuration; and the third, *Reid + Suhl-Suhl update*, attempts to find a triangular permutation of U by using the dense method proposed as a first step by Reid [20]. Because we can instrument our code, we have more details here:

- The columns labeled *%factor* indicate, for each instance, the percentage of time spent computing or updating the basis factorization. It is broken down into
  - *%b*, the percentage of time spent building fresh LU factorizations. In *refactor*, *%factor* = *%b* and it is further broken down into
    - *%p*, the time taken to find pseudo-triangular parts of the basis matrix, and
    - *%G*, the time spent performing Gaussian elimination.
  - *%*Δ, the percentage of time spent looking for triangular permutation updates, and
  - *%S*, the time required to perform the Suhl-Suhl update.
- The columns labeled *%solve* indicate the percentage of time spend solving systems of the type $Bx = b$ or $B^T y = e$, given a factorization of B (i.e., the cost of computing a factorization of B is not included).

Whenever multiple columns are grouped under a single category (for example, $\%p$ and $\%G$ grouped under $\%b$), a column labeled $+$ indicates the total for the parent category (in the example, $\%b$).

As noted previously, enabling the factorization update yields a drop in time per iteration, from 4.003ms to 0.797ms. However, attempting to skip some of the Suhl-Suhl updates by finding triangular permutations with the first step of Reid's update [20] makes the time per iteration rise again, to 1.119ms.

Then, we answer two computational questions that arise from our developments.

Success rate of our algorithm. The first question is related more to the nature of LP bases than to our method specifically: How often can a spiked upper triangular factor be permuted into a triangular matrix? As mentioned in Section 3, we already answered this question using the naive method inspired by Reid's update. On average, 53.897% of the spiked matrices can be permuted into a triangular one. We give more detailed results, obtained with our new method, in Tables 7 and 8. At each iteration, a basis factorization is needed. If we have one from the previous iteration, then we compute the spiked upper triangular factor W^k. The columns labeled *permute* indicate the number of iterations for which there exists a triangular permutation of W^k, and such a permutation is found by our method (*it* is the raw number of times this happens, and *%* gives that number as a percentage of the overall number of iterations). When this fails, we fall back on the Suhl-Suhl update method (columns labeled *Suhl-Suhl*). When this fails too, we compute from scratch a fresh LU factorization of the basis matrix (*refactor*). These three possibilities sum up to roughly 100%, but not necessarily exactly 100%, as the simplex code can decide to compute a fresh factorization, even after an update was successfully computed.

Note that both update methods (*permute* and *Suhl-Suhl*) may fail for an additional reason, besides the obvious ones (no previous factorization exists, or no triangular permutation exists). We enforce a Markowitz-type condition on the U factor: the diagonal elements of U must not be too small compared to the other elements in their row. In practice, we impose $|U_{ii}| > 0.001 \cdot \max_j |U_{ij}|$ for all i. Whenever an update would yield a U factor that would violate this condition, we abort the attempt. For a new factorization, the Markowitz threshold is 0.01.

Tables 7 and 8 indicate that our method succeeded in permuting W^k into a triangular matrix that satisfied the Markowitz condition in 56.354% of the iterations, on average over MIPLIB 2010 instances. This number differs slightly from the one mentioned previously (53.897% with Reid's update) simply because of the varying solution paths encountered. In every such occurrence, there is no need to perform Gaussian elimination and add an η-matrix to the factorization. In most of the remaining cases (43.207%), a Suhl-Suhl update is performed. With the Mittelmann test set, the percentages are 42.124% and 54.321%, respectively.

Computational cost of the algorithm. The second question is: Does our method make the dual simplex method faster overall? Table 4 summarizes the results of our experiments (details on Tables 9 and 10). It suggests that with our method enabled, we can solve MIPLIB 2010 problems around 5% faster, and Mittelmann problems around 14% faster, in geometric mean. If we consider the time per iteration, to account for varying iteration counts, the improvements become 2% and 6%, respectively. For most MIPLIB 2010 instances, 2% is

Factorization update method	it (geometric mean)	time (s) (geometric mean, shift = 10s)	t/it (ms) (geometric mean, shift = 10ms)	%Δ (avg)	%S (avg)	%[] (avg)	%solve (avg)
Miplib 2010:							
Suhl-Suhl	1229.210	4.014	0.797	0.00%	7.55%	0.34%	50.75%
permute + S.-S.	1217.246	3.830	0.778	0.75%	7.12%	0.36%	49.86%
Mittelmann:							
Suhl-Suhl	72568.431	630.194	14.444	0.00%	4.03%	0.08%	38.29%
permute + S.-S.	66512.027	540.739	13.535	0.13%	3.72%	0.06%	38.91%

Table 4: Impact of the permutation method on running time.

below the relative standard deviation in running time from one run to another, and variations in the update algorithms yield differing solution paths, creating even more noise in our measurements. Therefore, while our method seems beneficial, it is difficult to conclude it, from these numbers, with absolute certainty. However, the subsequent columns give us more information. The average percentage of time spent looking for triangular permutations of W^k (Pseudocode 2) is indicated in *%Δ*, and the time spent applying the resulting pivots (Figure 14) is indicated in *%[]*. As a comparison, *%S* denotes the Suhl-Suhl update, and *%solve* the triangular solves. On MIPLIB 2010, only 0.75% of the solution time was dedicated to looking for a triangular permutation of W^k (recall that one was found in 56.354% of the cases), and 0.36% to applying the necessary pivots. The numbers are even lower for Mittelmann problems (0.13% and 0.06%, respectively). We can thus be confident that even when the gains we obtain (from avoiding Suhl-Suhl updates and lowering the number of η-matrices) are small, they come with essentially negligible costs.

6 Conclusion

We present a new method for finding triangular permutations of spiked upper triangular matrices. It finds a permutation if and only if one exists. While other methods have been presented previously for the same task, our approach takes into account the sparsity of the matrix and performs only the pivots that are absolutely necessary to build the permutation.

Surprisingly, our experiments show that around half of the spiked upper triangular matrices can be permuted into a triangular one. Exploiting this fact only leads to modest gains in solving linear optimization problems, but we show that it has no significant drawbacks.

References

[1] Richard H. Bartels and Gene H. Golub. The simplex method of linear programming using LU decomposition. *Commun. ACM*, 12(5):266–268, May 1969.

[2] Robert E. Bixby. Solving LPs in practice, 2009. Communication at the Combinatorial Optimization at Work summer school, September 24th, 2009 `http://co-at-work.zib.de/berlin2009/downloads/2009-09-24/2009-09-24-1100-BB-Linear-Programming-2.pdf`.

[3] Vašek Chvátal. *Linear Programming.* W. H. Freeman and Company, New York, 1983.

[4] J.J.H. Forrest and J.A. Tomlin. Updated triangular factors of the basis to maintain sparsity in the product form simplex method. *Mathematical Programming*, 2(1):263–278, 1972.

[5] John Forrest, David de la Nuez, and Robin Lougee-Heimer. Coin-Clp user guide, 2004. `http://www.coin-or.org/Clp/userguide/index.html`.

[6] John J. Forrest and Donald Goldfarb. Steepest-edge simplex algorithms for linear programming. *Mathematical Programming*, 57(1):341–374, 1992.

[7] Robert Fourer. Notes on the dual simplex method. Draft report, 1994.

[8] Gurobi Optimization, Inc. Gurobi optimizer reference manual, 2014.

[9] J.A.J. Hall. *Sparse matrix algebra for active set methods in linear programming.* PhD thesis, University of Dundee Department of Mathematics and Computer Science, 1991.

[10] Qi Huangfu. *High performance simplex solver.* PhD thesis, University of Edinburgh, 2013.

[11] Qi Huangfu and J.A.Julian Hall. Novel update techniques for the revised simplex method. *Computational Optimization and Applications*, pages 1–22, 2014.

[12] F.M. Kirillova, R. Gabasov, and O.I. Kostyukova. A method of solving general linear programming problems. *Doklady AN BSSR*, 23(3):197–200, 1979. (in Russian).

[13] Achim Koberstein. *The Dual Simplex Method, Techniques for a fast and stable implementation.* PhD thesis, Fakultät für Wirtschaftswissenschaften der Universität Paderborn, 2005.

[14] Achim Koberstein. Progress in the dual simplex algorithm for solving large scale lp problems: techniques for a fast and stable implementation. *Computational Optimization and Applications*, 41(2):185–204, 2008.

[15] Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E. Bixby, Emilie Danna, Gerald Gamrath, Ambros M. Gleixner, Stefan Heinz, Andrea Lodi, Hans Mittelmann, Ted Ralphs, Domenico Salvagnin, Daniel E. Steffy, and Kati Wolter. MIPLIB 2010. *Mathematical Programming Computation*, 3(2):103–163, 2011.

[16] R. Luce, J. Duintjer Tebbens, Liesen, R. Nabben, M. Grötschel, T. Koch, and O. Schenk. On the factorization of simplex basis matrices. *ACM Transactions on Mathematical Software.* ZIB-Report 09-24 (July 2009).

[17] István Maros. *Computational Techniques of the Simplex Method.* Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[18] Hans Mittelmann. Benchmarks for optimization software, 2016. `http://plato.asu.edu/bench.html`.

[19] William Orchard-Hayes. *Advanced linear-programming computing techniques.* McGraw-Hill, New York, 1968.

[20] J.K. Reid. A sparsity-exploiting variant of the Bartels—Golub decomposition for linear programming bases. *Mathematical Programming*, 24(1):55–69, 1982.

[21] M.A. Saunders. *The complexity of computational problem solving*, chapter The complexity of LU updating in the simplex method, pages 214–230. University of Queensland Press, St. Lucia, Queensland, 1976.

[22] M.A. Saunders. *Sparse Matrix Computations*, chapter A fast, stable implementation of the simplex method using Bartels-Golub updating, pages 213–226. Academic Press, New York, 1976.

[23] Leena M. Suhl and Uwe H. Suhl. A fast LU update for linear programming. *Annals of Operations Research*, 43(1):33–47, 1993.

[24] Uwe H. Suhl and Leena M. Suhl. Computing sparse LU factorizations for large-scale linear programming bases. *ORSA Journal on Computing*, 2(4):325–335, 1990.

[25] The International Business Machines Corporation. IBM ILOG CPLEX Optimizer, 2014.

[26] Roland Wunderling. *Paralleler und objektorientierter Simplex-Algorithmus.* PhD thesis, Technische Universität Berlin, 1996. `http://www.zib.de/Publications/abstracts/TR-96-09/`.

[27] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic Discrete Methods*, 2(1):77–79, 1981.

instance	CPLEX (refactor) it	time (s)	t/it (ms)	CPLEX (update) it	time (s)	t/it (ms)
30n20b8	1743	0.647	0.371	2447	0.200	0.082
acc-tight5	2045	1.793	0.877	2099	0.290	0.138
aflow40b	1303	0.240	0.184	1282	0.050	0.039
air04	3783	3.317	0.877	3795	0.543	0.143
app1-2	894	3.827	4.280	860	0.533	0.620
ash608gpia-3col	7944	37.250	4.689	7456	4.787	0.642
bab5	16672	20.813	1.248	17611	0.997	0.057
beasleyC3	712	0.110	0.154	729	0.040	0.055
biella1	6336	9.673	1.527	6618	1.250	0.189
bienst2	119	0.017	0.140	119	0.000	0.000
binkar10_1	764	0.123	0.161	769	0.017	0.022
bley_xl1	44774	3600.040	80.405	178179	3600.083	20.205
bnatt350	930	0.597	0.642	932	0.110	0.118
core2536-691	16971	18.830	1.110	15404	2.323	0.151
cov1075	549	0.383	0.698	521	0.053	0.102
csched010	1975	0.293	0.149	1604	0.063	0.039
danoint	650	0.163	0.251	820	0.043	0.053
dfn-gwin-UUM	487	0.043	0.089	521	0.017	0.032
eil33-2	162	0.043	0.267	184	0.030	0.163
eilB101	515	0.130	0.252	447	0.047	0.104
enlight13	1	0.000	0.000	1	0.000	0.000
enlight14	1	0.000	0.000	1	0.000	0.000
ex9	20579	3600.333	174.952	340484	1977.483	5.808
glass4	36	0.000	0.000	36	0.000	0.000
gmu-35-40	300	0.050	0.167	294	0.010	0.034
iis-100-0-cov	385	0.460	1.195	383	0.090	0.235
iis-bupa-cov	867	1.483	1.711	915	0.240	0.262
iis-pima-cov	782	1.970	2.519	703	0.233	0.332
lectsched-4-obj	882	0.643	0.729	863	0.027	0.031
m100n500k4r1	429	0.047	0.109	365	0.017	0.046
macrophage	704	0.260	0.369	695	0.020	0.029
map18	28705	145.033	5.053	22809	3.227	0.141
map20	21943	109.020	4.968	16852	2.213	0.131
mcsched	3670	1.280	0.349	3349	0.200	0.060
mik-250-1-100-1	102	0.003	0.033	102	0.000	0.000
mine-166-5	1219	1.600	1.313	1199	0.187	0.156
mine-90-10	1837	1.777	0.967	1588	0.200	0.126
msc98-ip	27314	116.143	4.252	15030	4.360	0.290
mspp16	45	4.253	94.519	47	2.113	44.965
mzzv11	26321	88.717	3.371	20779	9.043	0.435
n3div36	299	0.257	0.858	477	0.150	0.314
n3seq24	3713	25.243	6.799	3406	4.800	1.409
n4-3	928	0.223	0.241	813	0.050	0.062
neos-1109824	139	0.133	0.959	138	0.010	0.072
neos-1337307	3577	3.997	1.117	3861	0.283	0.073
neos-1396125	2158	0.793	0.368	2332	0.150	0.064
neos-1601936	9905	25.253	2.550	10387	3.067	0.295
neos-476283	14277	185.310	12.980	9615	13.067	1.359
neos-686190	160	0.063	0.396	151	0.017	0.110
neos-849702	2453	2.223	0.906	2309	0.277	0.120
neos-916792	677	0.373	0.551	677	0.140	0.207
neos-934278	18451	52.107	2.824	16025	6.543	0.408
neos13	486	1.467	3.018	489	0.087	0.177
neos18	997	0.443	0.445	1047	0.087	0.083
net12	6028	16.303	2.705	5836	1.370	0.235
netdiversion	22823	415.517	18.206	20377	6.863	0.337
newdano	119	0.017	0.140	119	0.000	0.000
noswot	64	0.003	0.052	64	0.000	0.000
ns1208400	4560	7.840	1.719	4978	1.133	0.228
ns1688347	1238	1.147	0.926	1418	0.207	0.146
ns1758913	48340	2474.690	51.193	67596	131.643	1.948
ns1766074	44	0.000	0.000	44	0.000	0.000
ns1830653	977	0.753	0.771	1037	0.133	0.129
opm2-z7-s2	5069	25.250	4.981	5459	2.833	0.519
pg5_34	335	0.037	0.109	328	0.007	0.020
pigeon-10	223	0.030	0.135	212	0.010	0.047
pw-myciel4	1489	0.550	0.369	1575	0.120	0.076
qiu	1033	0.227	0.219	1057	0.037	0.035
rail507	3444	2.467	0.716	3792	0.763	0.201
ran16x16	333	0.013	0.040	328	0.010	0.030
reblock67	1008	0.497	0.493	1027	0.077	0.075
rmatr100-p10	1476	1.233	0.836	1940	0.237	0.122
rmatr100-p5	2719	2.710	0.997	3556	0.467	0.131
rmine6	1196	1.113	0.931	1223	0.140	0.114
rocII-4-11	243	0.240	0.988	246	0.030	0.122
rococoC10-001000	1278	0.193	0.151	1363	0.057	0.042
roll3000	650	0.260	0.400	759	0.047	0.061
satellites1-25	5702	10.050	1.763	5012	1.177	0.235
sp98ic	642	0.370	0.576	667	0.167	0.250
sp98ir	686	0.280	0.408	843	0.087	0.103
tanglegram1	333	3.203	9.620	321	0.290	0.903
tanglegram2	165	0.207	1.253	172	0.040	0.233
timtab1	21	0.000	0.000	21	0.000	0.000
triptim1	32649	340.553	10.431	31699	27.760	0.876
unitcal_7	19643	89.690	4.566	19986	0.643	0.032
vpphard	7213	38.917	5.395	6172	7.027	1.138
zib54-UUE	3447	1.613	0.468	3573	0.210	0.059
average	5504	132.244	6.328	10499	66.936	1.026
geom. mean	1166	9.598	2.749	1220	2.894	0.595

Table 5: CPLEX running time

instance	refactor							Suhl-Suhl update							Reid + Suhl-Suhl update							
	it	time (s)	t/it (ms)	%b			%solve	it	time (s)	t/it (ms)	%factor			%solve	it	time (s)	t/it (ms)	%factor				%solve
				%p	%G	+					%b	%S	+					%b	%Δ	%S	+	
30n20b8	2330	0.971	0.417	11	28	47	6	2160	0.207	0.096	3	3	7	29	2268	0.238	0.105	2	11	3	18	28
acc-tight5	2359	3.448	1.462	14	56	79	6	1720	0.478	0.278	2	20	24	57	1735	0.707	0.408	1	27	14	46	41
aflow40b	1493	0.400	0.268	31	0	48	8	1576	0.085	0.054	1	4	8	63	1465	0.119	0.081	1	31	2	41	35
air04	2827	2.771	0.980	8	45	58	5	3013	0.648	0.215	2	6	10	30	2964	0.715	0.241	2	11	5	20	26
app1-2	1041	6.838	6.569	40	0	59	10	1035	1.238	1.197	2	4	9	57	1029	1.795	1.745	1	31	3	37	41
ash608gpia-3col	6410	49.741	7.760	28	35	76	4	6878	19.396	2.820	1	23	24	53	8018	35.684	4.451	0	33	13	47	36
bab5	16560	41.684	2.517	23	37	68	3	14003	2.806	0.200	1	11	13	60	14018	5.087	0.363	0	40	7	51	34
beasleyC3	835	0.192	0.230	31	0	47	12	845	0.067	0.079	1	7	11	65	851	0.076	0.089	1	30	0	38	42
biella1	6920	12.167	1.758	12	46	66	6	7039	2.096	0.298	3	6	10	42	6447	2.346	0.364	2	15	5	23	37
bienst2	231	0.030	0.128	28	9	54	9	409	0.021	0.051	4	4	12	58	365	0.012	0.034	3	19	4	32	45
binkar10_1	1048	0.172	0.164	32	0	49	3	1031	0.019	0.019	3	3	10	48	1029	0.026	0.026	2	31	3	44	27
bley_xl1	18536	3600.110	194.226	15	65	87	4	116684	2018.235	17.297	6	2	8	60	55369	1357.463	24.517	8	32	1	42	38
bnatt350	1248	0.659	0.528	38	1	60	4	1049	0.251	0.239	2	18	21	62	1013	0.252	0.249	1	36	6	47	40
core2536-691	21738	57.695	2.654	12	53	70	5	18536	6.801	0.367	1	6	9	49	22825	11.367	0.498	1	17	4	23	41
cov1075	440	0.286	0.650	20	47	77	8	647	0.109	0.168	7	14	24	56	673	0.138	0.204	5	20	12	39	47
csched010	2485	0.446	0.179	14	28	50	8	2369	0.113	0.048	3	5	10	42	2220	0.129	0.058	2	15	5	24	38
danoint	857	0.190	0.222	20	31	63	10	829	0.053	0.064	3	10	16	55	720	0.075	0.104	2	17	8	30	45
dfn-gwin-UUM	517	0.040	0.078	18	13	40	9	545	0.012	0.021	5	4	12	34	531	0.022	0.042	4	13	3	23	29
eil33-2	173	0.064	0.368	2	7	10	2	175	0.030	0.171	3	1	4	3	192	0.045	0.235	3	1	1	5	3
eilB101	435	0.109	0.252	5	25	35	4	459	0.054	0.117	5	2	8	12	459	0.058	0.127	4	4	2	11	11
enlight13	3	0.000	0.057	25	0	41	8	5	0.000	0.057	34	2	44	11	5	0.000	0.046	42	1	0	44	10
enlight14	3	0.000	0.152	27	0	42	8	5	0.000	0.066	35	3	43	10	5	0.000	0.065	41	1	0	43	10
ex9	11563	3600.338	311.358	2	94	97	1	70143	1426.987	20.344	52	7	59	27	71358	1981.369	27.767	54	16	3	73	17
glass4	72	0.004	0.056	32	0	52	2	72	0.001	0.010	25	2	32	14	72	0.001	0.015	29	5	0	37	11
gmu-35-40	198	0.028	0.139	25	1	40	11	208	0.008	0.039	5	3	12	51	190	0.007	0.037	4	19	2	31	36
iis-100-0-cov	347	0.477	1.375	38	8	65	11	342	0.176	0.515	2	21	25	58	350	0.227	0.649	2	26	14	44	44
iis-bupa-cov	822	1.615	1.965	37	12	66	11	790	0.352	0.445	3	12	17	63	817	0.539	0.660	2	28	10	42	44
iis-pima-cov	824	2.202	2.672	40	5	64	12	788	0.437	0.555	3	6	12	64	814	0.746	0.917	2	31	7	42	43
lectsched-4-obj	1225	0.898	0.733	39	0	59	2	951	0.090	0.094	4	1	7	65	950	0.187	0.197	2	49	0	61	27
m100n500k4r1	553	0.065	0.118	7	50	64	8	490	0.026	0.052	11	7	21	34	470	0.026	0.054	10	10	6	29	31
macrophage	727	0.374	0.514	42	1	63	2	725	0.023	0.032	11	1	14	33	725	0.080	0.111	3	60	0	76	9
map18	14810	109.168	7.371	50	1	66	6	13463	9.894	0.735	1	1	2	86	14134	24.079	1.704	0	46	0	51	43
map20	14509	106.739	7.357	50	1	67	6	13062	8.932	0.684	1	1	2	86	12809	17.789	1.389	0	45	0	51	43
mcsched	3509	1.674	0.477	33	9	57	14	3602	0.526	0.146	1	12	14	66	3435	0.697	0.203	1	26	11	40	47
mik-250-1-100-1	102	0.003	0.031	16	0	26	3	102	0.001	0.009	8	3	18	17	102	0.001	0.008	11	3	0	19	17
mine-166-5	1161	2.080	1.792	41	4	64	9	1123	0.449	0.399	1	23	27	56	1151	1.042	0.905	1	42	14	62	30
mine-90-10	1864	3.497	1.876	28	31	72	8	1717	0.746	0.434	1	24	28	58	1686	1.440	0.854	1	36	16	56	36
msc98-ip	19188	159.128	8.293	28	36	75	6	22552	30.655	1.359	1	10	12	69	17344	30.250	1.744	1	41	4	48	42
mspp16	41	6.552	159.813	16	0	27	0	51	0.500	9.812	28	0	29	1	51	0.544	10.671	30	3	0	33	1
mzzv11	25220	252.021	9.993	13	64	83	5	30077	32.721	1.088	2	7	9	64	10293	10.929	1.062	1	37	5	45	40
n3div36	201	0.266	1.324	15	1	24	1	196	0.082	0.417	2	0	2	5	191	0.086	0.451	2	4	0	7	4
n3seq24	2896	19.352	6.682	5	1	8	1	3704	12.615	3.406	0	0	1	5	3626	12.939	3.569	0	2	0	3	4
n4-3	990	0.318	0.322	26	7	46	8	1018	0.096	0.095	1	7	10	45	907	0.117	0.129	1	27	9	40	31
neos-1109824	150	0.207	1.382	39	1	61	5	141	0.026	0.186	18	1	21	44	141	0.065	0.461	7	51	0	72	15
neos-1337307	4111	6.930	1.686	39	14	69	7	3840	1.083	0.282	1	19	23	61	4097	1.941	0.474	1	40	6	52	37
neos-1396125	2516	1.243	0.494	25	26	63	13	2348	0.289	0.123	2	10	13	66	2249	0.359	0.160	2	27	6	38	46
neos-1601936	9251	45.032	4.868	10	66	83	4	4087	1.939	0.474	2	12	16	51	3956	2.544	0.643	2	19	10	33	42
neos-476283	9802	134.041	13.675	26	9	47	5	10258	14.534	1.417	7	1	9	52	9254	18.036	1.949	5	25	1	33	37
neos-686190	266	0.158	0.593	31	2	51	7	295	0.043	0.145	5	6	13	47	296	0.045	0.152	3	30	3	45	29
neos-849702	3378	5.124	1.517	8	67	81	5	1344	0.288	0.214	2	17	21	50	1417	0.351	0.248	2	18	14	36	41
neos-916792	552	0.390	0.707	20	14	45	10	552	0.091	0.166	3	8	15	45	552	0.108	0.195	3	15	6	28	38
neos-934278	17366	488.826	28.148	4	89	95	1	17881	33.288	1.862	3	20	24	54	17794	42.078	2.365	3	25	13	42	40
neos13	395	1.696	4.294	40	0	61	8	389	0.167	0.430	7	0	19	65	389	0.162	0.416	7	6	0	15	69
neos18	1346	0.769	0.571	41	2	63	8	1183	0.154	0.130	1	13	16	65	1173	0.213	0.182	1	50	4	62	29
net12	6521	26.576	4.075	37	11	64	10	3366	2.387	0.709	1	4	6	79	4397	4.859	1.105	0	29	4	35	54
netdiversion	43386	3600.084	82.979	15	69	89	1	45217	149.291	3.302	0	2	3	91	45111	442.350	9.806	0	58	0	63	35
newdano	231	0.039	0.167	28	9	54	9	409	0.015	0.038	4	4	12	58	365	0.026	0.070	3	20	4	32	45
noswot	40	0.002	0.052	28	1	45	9	42	0.001	0.019	20	3	30	33	42	0.001	0.023	20	13	1	41	28
ns1208400	5147	11.775	2.288	15	52	75	6	2080	0.624	0.300	2	13	17	53	4547	2.091	0.460	3	19	8	32	43
ns1688347	709	0.582	0.821	28	14	57	9	1282	0.496	0.387	2	17	22	61	1264	0.558	0.442	2	37	7	50	35
ns1758913	4918	34.798	7.076	35	7	58	6	12069	21.489	1.781	1	8	10	73	10449	30.165	2.887	1	41	3	47	43
ns1766074	33	0.001	0.029	25	0	42	9	32	0.001	0.018	21	3	33	27	32	0.001	0.022	18	19	0	48	20
ns1830653	1350	1.096	0.812	28	21	65	11	1101	0.231	0.209	2	13	18	64	1343	0.360	0.268	1	23	10	37	49
opm2-z7-s2	3921	33.772	8.613	44	9	68	10	3783	10.215	2.700	1	26	29	55	3989	29.182	7.316	0	39	18	60	34
pg5_34	334	0.044	0.131	13	0	20	5	307	0.008	0.027	2	1	7	45	307	0.006	0.020	2	15	0	22	31
pigeon-10	311	0.050	0.162	35	3	56	12	264	0.011	0.043	7	3	13	66	266	0.016	0.061	4	29	2	39	47
pw-myciel4	1387	0.865	0.624	16	48	72	8	1377	0.199	0.145	2	15	19	59	1315	0.225	0.171	1	22	12	38	45
qiu	1089	0.296	0.272	30	20	66	7	1113	0.048	0.043	4	5	10	67	1118	0.101	0.090	3	37	8	53	37
rail507	3474	4.058	1.168	6	17	26	3	3180	1.249	0.393	1	2	3	11	3039	1.272	0.418	1	4	2	7	11
ran16x16	327	0.028	0.086	27	0	43	10	331	0.006	0.019	4	3	12	61	331	0.004	0.012	4	23	0	33	39
reblock67	1019	1.045	1.026	19	50	79	6	1018	0.173	0.170	2	19	23	61	1027	0.251	0.244	2	39	9	55	35
rmatr100-p10	1181	1.629	1.380	34	6	55	12	1183	0.362	0.306	2	4	7	72	1249	0.508	0.407	1	24	4	32	53
rmatr100-p5	2372	4.209	1.775	33	9	56	13	2365	0.925	0.391	1	5	7	72	2323	1.163	0.500	1	25	3	32	52
rmine6	1202	1.620	1.348	39	7	64	8	1167	0.280	0.240	1	21	24	58	1179	0.572	0.485	1	54	7	68	24
rocII-4-11	362	0.445	1.230	39	1	61	7	426	0.061	0.144	9	2	15	58	422	0.111	0.264	5	48	1	61	26
rococoC10-001000	1086	0.188	0.173	24	9	45	10	1053	0.040	0.038	2	8	12	60	1018	0.071	0.070	1	29	5	40	43
roll3000	897	0.368	0.410	33	13	61	10	948	0.089	0.094	3	8	14	66	862	0.115	0.133	2	35	6	47	41
satellites1-25	10886	69.090	6.347	10	72	86	4	5040	4.185	0.830	1	20	23	57	4712	4.718	1.001	1	27	11	41	41
sp98ic	557	0.552	0.990	7	5	16	3	577	0.170	0.295	1	2	4	13	485	0.137	0.282	1	4	2	8	12
sp98ir	524	0.263	0.501	26	11	49	10	561	0.092	0.165	3	7	12	50	529	0.113	0.213	2	23	7	37	37
tanglegram1	518	7.236	13.969	36	0	59	10	362	0.620	1.713	6	0	7	70	373	1.071	2.871	3	29	0	39	45
tanglegram2	172	0.271	1.573	35	0	57	14	184	0.064	0.347	10	0	12	69	184	0.074	0.403	8	23	0	38	49
timtab1	177	0.010	0.055	25	0	40	6	178	0.002	0.011	9	3	18	36	178	0.001	0.007	9	11	0	28	30
triptim1	47743	3051.189	63.909	7	85	94	1	43642	112.778	2.584	4	10	14	62	46390	185.256	3.993	3	34	5	43	43
unitcal_7	15500	104.492	6.741	47	0	65	1	15686	1.357	0.086	10	0	12	71	15503	7.601	0.490	2	72	0	83	13
vpphard	4478	39.413	8.802	26	24	63	7	1583	2.354	1.487	1	10	13	57	1450	3.408	2.350	1	31	7	42	40
zib54-UUE	4963	3.550	0.715	23	31	63	8	4492	0.456	0.101	5	3	9	60	4178	0.512	0.123	3	20	4	29	47
average	4658	180.792	11.771	25	20	58	7	6263	45.290	1.025	6	8	15	51	5367	49.221	1.510	5	26	5	39	34
geom. mean	1213	11.853	4.003					1229	4.014	0.797					1203	4.880	1.119					

Table 6: Running time: refactor, Suhl-Suhl, Reid

instance	it	refactor		permute		Suhl-Suhl	
		it	%	it	%	it	%
30n20b8	2457	23	0.94	939	38.22	1499	61.01
acc-tight5	1682	8	0.48	627	37.28	1051	62.49
aflow40b	1465	3	0.20	1428	97.47	37	2.53
air04	3043	20	0.66	812	26.68	2213	72.72
app1-2	1047	5	0.48	310	29.61	735	70.20
ash608gpia-3col	7634	10	0.13	2927	38.34	4703	61.61
bab5	13667	11	0.08	10643	77.87	3022	22.11
beasleyC3	851	3	0.35	851	100.00	0	0.00
biella1	6684	36	0.54	1855	27.75	4794	71.72
bienst2	351	3	0.85	229	65.24	122	34.76
binkar10_1	1035	3	0.29	997	96.33	38	3.67
bley_xl1	85616	1709	2.00	8482	9.91	75430	88.10
bnatt350	1111	5	0.45	508	45.72	602	54.19
core2536-691	20361	35	0.17	10407	51.11	9921	48.73
cov1075	578	7	1.21	16	2.77	558	96.54
csched010	2419	21	0.87	1199	49.57	1203	49.73
danoint	867	10	1.15	203	23.41	657	75.78
dfn-gwin-UUM	546	13	2.38	288	52.75	248	45.42
eil33-2	192	22	11.46	17	8.85	156	81.25
eilB101	459	19	4.14	42	9.15	401	87.36
enlight13	5	3	60.00	5	100.00	0	0.00
enlight14	5	3	60.00	5	100.00	0	0.00
ex9	70972	1265	1.78	3232	4.55	66482	93.67
glass4	72	3	4.17	72	100.00	0	0.00
gmu-35-40	194	3	1.55	164	84.54	30	15.46
iis-100-0-cov	328	3	0.91	29	8.84	299	91.16
iis-bupa-cov	719	5	0.70	51	7.09	666	92.63
iis-pima-cov	862	7	0.81	46	5.34	812	94.20
lectsched-4-obj	950	4	0.42	932	98.11	18	1.89
m100n500k4r1	489	17	3.48	114	23.31	361	73.82
macrophage	725	4	0.55	643	88.69	82	11.31
map18	14128	11	0.08	12122	85.80	2005	14.19
map20	13184	10	0.08	11559	87.67	1625	12.33
mcsched	3587	13	0.36	1641	45.75	1936	53.97
mik-250-1-100-1	102	2	1.96	102	100.00	0	0.00
mine-166-5	1144	2	0.17	803	70.19	341	29.81
mine-90-10	1651	4	0.24	929	56.27	720	43.61
msc98-ip	20710	40	0.19	8032	38.78	12659	61.13
mspp16	51	3	5.88	51	100.00	0	0.00
mzzv11	33250	105	0.32	11495	34.57	21652	65.12
n3div36	232	3	1.29	54	23.28	177	76.29
n3seq24	3389	12	0.35	1742	51.40	1639	48.36
n4-3	919	5	0.54	732	79.65	185	20.13
neos-1109824	143	3	2.10	92	64.34	51	35.66
neos-1337307	3837	8	0.21	3042	79.28	791	20.62
neos-1396125	2830	21	0.74	1170	41.34	1643	58.06
neos-1601936	3682	17	0.46	654	17.76	3014	81.86
neos-476283	8037	147	1.83	3258	40.54	4635	57.67
neos-686190	301	3	1.00	128	42.52	173	57.48
neos-849702	1450	9	0.62	294	20.28	1151	79.38
neos-916792	552	10	1.81	84	15.22	461	83.51
neos-934278	18272	35	0.19	6573	35.97	11667	63.85
neos13	389	4	1.03	385	98.97	4	1.03
neos18	1168	4	0.34	863	73.89	305	26.11
net12	4447	6	0.13	2576	57.93	1869	42.03
netdiversion	44148	26	0.06	35399	80.18	8749	19.82
newdano	351	3	0.85	229	65.24	122	34.76
noswot	42	3	7.14	37	88.10	5	11.90
ns1208400	2256	15	0.66	360	15.96	1885	83.55
ns1688347	1220	4	0.33	649	53.20	571	46.80
ns1758913	8341	15	0.18	4582	54.93	3749	44.95
ns1766074	32	3	9.38	32	100.00	0	0.00
ns1830653	980	5	0.51	245	25.00	733	74.80
opm2-z7-s2	4176	5	0.12	2463	58.98	1711	40.97
pg5_34	307	2	0.65	307	100.00	0	0.00
pigeon-10	266	4	1.50	215	80.83	51	19.17
pw-myciel4	1235	7	0.57	405	32.79	827	66.96
qiu	1311	4	0.31	991	75.59	319	24.33
rail507	2940	18	0.61	1201	40.85	1724	58.64
ran16x16	331	3	0.91	331	100.00	0	0.00
reblock67	1051	3	0.29	521	49.57	529	50.33
rmatr100-p10	1226	6	0.49	523	42.66	700	57.10
rmatr100-p5	2318	8	0.35	987	42.58	1326	57.20
rmine6	1160	2	0.17	750	64.66	410	35.34
rocII-4-11	421	4	0.95	295	70.07	126	29.93
rococoC10-001000	1112	5	0.45	865	77.79	246	22.12
roll3000	945	4	0.42	494	52.28	450	47.62
satellites1-25	4573	10	0.22	2233	48.83	2334	51.04
sp98ic	591	9	1.52	136	23.01	451	76.31
sp98ir	516	4	0.78	244	47.29	271	52.52
tanglegram1	369	4	1.08	345	93.50	24	6.50
tanglegram2	184	4	2.17	176	95.65	8	4.35
timtab1	178	3	1.69	178	100.00	0	0.00
triptim1	44006	64	0.15	15847	36.01	28098	63.85
unitcal_7	15502	25	0.16	14729	95.01	756	4.88
vpphard	1510	5	0.33	538	35.63	971	64.30
zib54-UUE	3757	30	0.80	3256	86.66	475	12.64
average			2.54		56.35		43.21

Table 7: Use of each update method (Miplib 2010 preprocessed)

instance	it	refactor		permute		Suhl-Suhl	
		it	%	it	%	it	%
L1_sixm250obs	86485	44	0.05	79897	92.38	6588	7.62
Linf_520c	32466	589	1.81	19218	59.19	12671	39.03
buildingenergy	116101	63	0.05	107045	92.20	9055	7.80
cont1	45022	355	0.79	3468	7.70	41227	91.57
cont11	84525	137	0.16	40539	47.96	43893	51.93
cont4	40803	59	0.14	801	1.96	39965	97.95
dano3mip	24494	455	1.86	6556	26.77	17489	71.40
dbic1	116330	9726	8.36	76049	65.37	30562	26.27
dfl001	24518	62	0.25	10876	44.36	13584	55.40
ds-big	42179	552	1.31	8137	19.29	33461	79.33
fome12	97219	211	0.22	43373	44.61	53648	55.18
fome13	236861	546	0.23	99610	42.05	136735	57.73
gen4	866	15	1.73	8	0.92	845	97.58
ken-18	118434	63	0.05	117612	99.31	822	0.69
l30	11664	121	1.04	343	2.94	11206	96.07
lp22	20260	201	0.99	3853	19.02	16209	80.00
mod2	44185	302	0.68	25018	56.62	18865	42.70
neos	111552	62	0.06	78083	70.00	33460	29.99
neos1	47658	30	0.06	3396	7.13	44255	92.86
neos2	59135	49	0.08	2856	4.83	56245	95.11
neos3	34975	18	0.05	417	1.19	34558	98.81
ns1644855	69877	185	0.26	31633	45.27	38075	54.49
ns1687037	13357	1760	13.18	6914	51.76	4838	36.22
ns1688926	85364	84375	98.84	451	0.53	547	0.64
nug08-3rd	29543	100	0.34	6016	20.36	23433	79.32
nug15	552483	5917	1.07	23166	4.19	523402	94.74
pds-100	277868	152	0.05	242936	87.43	34898	12.56
pds-40	117622	77	0.07	93104	79.16	24481	20.81
qap12	126126	1490	1.18	7272	5.77	117368	93.06
qap15	550514	6309	1.15	24412	4.43	519797	94.42
rail4284	36523	122	0.33	13785	37.74	22617	61.93
self	57127	3254	5.70	4459	7.81	49423	86.51
sgpf5y6	256406	137	0.05	229990	89.70	26416	10.30
stat96v1	16534	151	0.91	2305	13.94	14084	85.18
stat96v4	39574	207	0.52	12180	30.78	27192	68.71
stormG2-125	101052	53	0.05	94900	93.91	6152	6.09
stormG2_1000	803485	404	0.05	758446	94.39	45039	5.61
stp3d	102296	88	0.09	77583	75.84	24656	24.10
watson_2	478328	340	0.07	380017	79.45	98154	20.52
world	49651	370	0.75	28140	56.68	21145	42.59
average			3.62		42.12		54.32

Table 8: Use of each update method (Mittelmann testset)

instance	Suhl-Suhl update								permute + Suhl-Suhl update								
	it	time (s)	t/it (ms)	%factor				%solve	it	time (s)	t/it (ms)	%factor					%solve
				%b	%S	%[]	+					%b	%Δ	%S	%[]	+	
30n20b8	2160	0.207	0.096	2.71	3.27	0.06	7.46	28.58	2457	0.213	0.087	2.63	0.62	3.16	0.06	7.83	29.05
acc-tight5	1720	0.478	0.278	1.84	19.55	0.05	23.77	57.20	1682	0.480	0.285	2.09	0.62	20.63	0.05	25.41	55.75
aflow40b	1576	0.085	0.054	1.02	3.72	0.14	7.60	62.92	1465	0.079	0.054	0.87	2.70	3.90	0.32	9.07	55.34
air04	3013	0.648	0.215	2.22	6.25	0.03	9.53	29.82	3043	0.649	0.213	2.29	0.43	6.33	0.03	10.08	29.26
app1-2	1035	1.238	1.197	1.52	4.20	0.15	8.88	57.48	1047	1.265	1.208	1.57	0.72	2.78	0.14	7.58	55.98
ash608gpia-3col	6878	19.396	2.820	0.56	22.70	0.04	24.08	53.21	7634	19.997	2.619	0.61	0.44	18.77	0.04	20.72	53.47
bab5	14003	2.806	0.200	0.78	11.10	0.04	13.04	59.64	13667	2.431	0.178	0.73	0.86	13.84	0.05	16.70	58.81
beasleyC3	845	0.067	0.079	1.35	6.86	0.17	10.84	65.08	851	0.062	0.073	1.14	4.46	0.00	0.57	6.80	63.11
biella1	7039	2.096	0.298	2.50	6.03	0.03	9.68	41.54	6684	1.971	0.295	2.73	0.38	6.30	0.03	10.65	41.77
bienst2	409	0.021	0.051	4.01	4.38	0.30	11.68	58.02	351	0.012	0.035	2.94	1.41	5.49	0.31	13.60	57.78
binkar10_1	1031	0.019	0.019	3.31	3.47	0.44	10.12	48.05	1035	0.017	0.017	3.36	2.00	5.23	0.59	13.74	43.56
bley_xl1	116684	2018.235	17.297	6.06	1.58	0.12	7.76	59.95	85616	1408.438	16.451	7.79	0.06	1.26	0.14	9.27	60.17
bnatt350	1049	0.251	0.239	1.58	18.05	0.10	21.21	62.15	1111	0.236	0.212	1.57	0.63	15.34	0.13	18.96	61.87
core2536-691	18536	6.801	0.367	1.06	6.40	0.02	8.63	49.33	20361	8.014	0.394	0.97	0.46	5.61	0.02	8.01	48.15
cov1075	647	0.109	0.168	7.43	13.91	0.08	24.45	56.49	578	0.100	0.172	6.89	0.65	14.96	0.08	25.75	55.79
csched010	2369	0.113	0.048	2.70	5.17	0.09	10.02	41.55	2419	0.137	0.057	2.30	1.00	4.98	0.11	10.74	44.92
danoint	829	0.053	0.064	2.97	9.76	0.12	15.78	55.25	867	0.066	0.076	3.80	1.15	7.72	0.14	15.40	52.53
dfn-gwin-UUM	545	0.012	0.021	4.86	4.01	0.25	11.54	34.05	546	0.019	0.035	4.07	1.71	4.25	0.32	13.09	34.72
eil33-2	175	0.030	0.171	2.65	0.57	0.04	3.84	3.13	192	0.038	0.197	2.83	0.17	0.52	0.04	4.17	2.98
eilB101	459	0.054	0.117	4.70	2.23	0.06	8.04	11.92	459	0.047	0.102	4.56	0.43	2.05	0.06	8.22	11.83
enlight13	5	0.000	0.057	33.94	2.33	3.03	43.50	10.85	5	0.000	0.055	35.62	1.59	0.00	2.68	43.06	9.39
enlight14	5	0.000	0.066	34.97	3.04	3.75	43.08	9.63	5	0.000	0.068	32.74	1.38	0.00	3.25	39.74	8.38
ex9	70143	1426.987	20.344	52.15	6.64	0.02	59.04	27.02	70972	1700.475	23.960	62.60	0.07	3.85	0.02	66.70	21.27
glass4	72	0.001	0.010	25.21	2.24	2.29	32.13	14.21	72	0.001	0.011	27.13	1.62	0.00	2.36	33.22	12.55
gmu-35-40	208	0.008	0.039	4.74	2.88	0.38	11.80	51.31	194	0.006	0.032	4.93	1.48	2.25	0.49	12.73	46.15
iis-100-0-cov	342	0.176	0.515	2.14	20.73	0.13	25.28	57.78	328	0.162	0.494	2.36	1.04	18.44	0.13	24.40	57.75
iis-bupa-cov	790	0.352	0.445	2.87	12.15	0.11	17.13	62.87	719	0.365	0.508	2.44	0.59	15.50	0.10	20.50	61.00
iis-pima-cov	788	0.437	0.555	3.33	6.40	0.14	12.00	63.88	862	0.478	0.555	3.28	0.53	7.82	0.13	13.68	63.02
lectsched-4-obj	951	0.090	0.094	4.03	0.93	0.28	6.55	64.75	950	0.058	0.061	4.70	0.27	0.02	0.31	6.31	63.65
m100n500k4r1	490	0.026	0.052	11.14	6.74	0.15	20.88	34.33	489	0.022	0.045	11.27	1.10	6.30	0.16	22.04	34.31
macrophage	725	0.023	0.032	10.60	0.62	0.76	13.98	32.99	725	0.022	0.030	10.77	0.46	0.13	0.79	14.17	31.42
map18	13463	9.894	0.735	0.67	0.95	0.04	2.13	85.52	14128	11.805	0.836	0.57	0.18	0.65	0.03	1.95	84.79
map20	13062	8.932	0.684	0.62	1.16	0.04	2.25	85.90	13184	8.197	0.622	0.66	0.24	0.63	0.03	1.96	86.26
mcsched	3602	0.526	0.146	1.06	11.77	0.06	13.79	66.09	3587	0.541	0.151	0.94	0.55	13.70	0.07	16.02	64.55
mik-250-1-100-1	102	0.001	0.009	7.87	3.22	1.10	18.12	17.32	102	0.001	0.006	8.37	1.14	0.00	1.20	16.37	17.13
mine-166-5	1123	0.449	0.399	1.18	23.43	0.09	26.67	55.62	1144	0.417	0.365	1.28	0.47	24.27	0.10	27.91	54.35
mine-90-10	1717	0.746	0.434	0.81	24.47	0.06	27.64	58.03	1651	0.571	0.346	1.01	0.55	16.02	0.07	19.53	62.02
msc98-ip	22552	30.655	1.359	1.47	9.54	0.04	11.94	69.21	20710	25.334	1.223	1.48	0.34	9.47	0.05	12.37	66.10
mspp16	51	0.500	9.812	27.79	0.00	2.33	28.90	1.14	51	0.485	9.506	28.47	0.00	0.00	2.36	29.24	1.16
mzzv11	30077	32.721	1.088	1.51	6.53	0.03	8.96	64.37	33250	32.193	0.968	2.34	0.36	5.50	0.04	9.11	60.63
n3div36	196	0.082	0.417	1.58	0.26	0.14	2.43	4.56	232	0.106	0.458	1.22	0.11	0.20	0.14	2.16	4.64
n3seq24	3704	12.615	3.406	0.09	0.41	0.01	0.67	4.50	3389	11.138	3.287	0.10	0.05	0.34	0.01	0.65	4.11
n4-3	1018	0.096	0.095	1.39	6.52	0.13	10.41	44.98	919	0.080	0.087	1.22	0.93	11.67	0.13	15.43	41.71
neos-1109824	141	0.026	0.186	18.39	0.51	1.56	21.03	43.72	143	0.025	0.178	18.42	0.38	0.57	1.55	21.54	42.73
neos-1337307	3840	1.083	0.282	1.49	19.20	0.12	23.07	60.99	3837	0.836	0.218	2.70	0.49	13.63	0.15	19.31	62.78
neos-1396125	2348	0.289	0.123	1.61	9.91	0.07	13.25	65.75	2830	0.314	0.111	2.67	0.71	8.29	0.08	13.14	64.05
neos-1601936	4087	1.939	0.474	2.40	11.85	0.04	15.76	51.03	3682	1.928	0.524	2.14	0.43	14.76	0.04	18.89	51.14
neos-476283	10258	14.534	1.417	6.50	1.40	0.08	9.39	51.91	8037	10.575	1.316	6.49	0.18	1.25	0.09	9.41	51.10
neos-686190	295	0.043	0.145	4.62	5.96	0.40	12.74	46.55	301	0.045	0.151	3.73	0.61	5.79	0.38	12.43	46.89
neos-849702	1344	0.288	0.214	2.40	16.82	0.05	21.15	49.61	1450	0.313	0.216	2.35	0.53	17.14	0.05	21.97	49.61
neos-916792	552	0.091	0.166	3.33	7.75	0.13	14.97	45.12	552	0.079	0.143	3.51	0.45	7.28	0.13	15.00	44.56
neos-934278	17881	33.288	1.862	2.86	19.96	0.02	23.97	53.77	18272	30.377	1.662	3.87	0.37	17.36	0.02	22.68	52.30
neos13	389	0.167	0.430	6.52	0.21	0.45	18.58	65.46	389	0.146	0.375	7.23	0.08	0.01	0.49	9.22	73.18
neos18	1183	0.154	0.130	1.45	13.37	0.14	15.94	64.63	1168	0.118	0.101	1.77	0.90	9.58	0.16	13.51	64.09
net12	3366	2.387	0.709	0.78	4.27	0.04	6.00	79.42	4447	3.651	0.821	0.55	0.33	5.65	0.03	7.25	78.67
netdiversion	45217	149.291	3.302	0.35	1.57	0.09	3.18	90.77	44148	142.726	3.233	0.37	0.16	0.90	0.07	2.61	91.14
newdano	409	0.015	0.038	3.99	4.39	0.27	11.66	58.23	351	0.016	0.045	2.88	1.38	5.41	0.30	13.44	57.65
noswot	42	0.001	0.019	20.05	2.69	1.38	30.33	32.56	42	0.001	0.021	20.84	1.92	1.59	1.66	31.47	30.44
ns1208400	2080	0.624	0.300	2.35	13.17	0.05	17.20	52.71	2256	0.760	0.337	2.52	0.43	13.47	0.05	18.37	52.31
ns1688347	1282	0.496	0.387	1.69	16.96	0.07	21.81	61.40	1220	0.424	0.348	1.36	0.48	16.94	0.06	21.70	61.61
ns1758913	12069	21.489	1.781	0.91	7.71	0.04	9.87	72.51	8341	11.446	1.372	1.24	0.42	5.96	0.04	8.68	72.54
ns1766074	32	0.001	0.018	21.36	3.21	1.61	32.72	26.54	32	0.000	0.011	21.82	1.88	0.00	1.79	31.94	24.70
ns1830653	1101	0.231	0.209	1.86	13.37	0.06	18.18	63.59	980	0.212	0.216	1.62	0.46	14.54	0.05	19.44	63.27
opm2-z7-s2	3783	10.215	2.700	0.82	26.34	0.05	29.05	54.77	4176	8.211	1.966	1.01	0.26	16.56	0.05	19.73	61.17
pg5_34	307	0.008	0.027	2.18	1.22	0.35	6.58	45.27	307	0.007	0.021	2.02	0.87	0.00	0.49	5.74	38.37
pigeon-10	264	0.011	0.043	7.00	2.72	0.35	13.18	66.25	266	0.013	0.050	6.04	1.20	2.65	0.37	12.82	66.55
pw-myciel4	1377	0.199	0.145	1.75	15.03	0.07	18.98	59.26	1235	0.179	0.145	1.62	0.65	17.19	0.08	21.57	57.85
qiu	1113	0.048	0.043	3.85	4.51	0.19	10.39	66.65	1311	0.080	0.061	2.61	1.42	18.65	0.12	24.22	62.08
rail507	3180	1.249	0.393	0.67	1.71	0.02	2.92	11.36	2940	1.169	0.397	0.56	0.24	1.67	0.01	3.05	11.60
ran16x16	331	0.006	0.019	3.63	3.31	0.42	11.69	61.14	331	0.005	0.015	4.03	3.28	0.00	0.87	11.28	51.81
reblock67	1018	0.173	0.170	2.29	18.52	0.09	22.86	61.25	1051	0.200	0.191	1.60	0.72	24.97	0.08	29.06	57.11
rmatr100-p10	1183	0.362	0.306	1.57	4.19	0.15	6.99	71.73	1226	0.361	0.294	1.80	0.33	4.58	0.14	7.78	70.69
rmatr100-p5	2365	0.925	0.391	1.40	5.09	0.12	7.18	71.82	2318	0.909	0.392	1.21	0.31	5.18	0.09	7.45	72.27
rmine6	1167	0.280	0.240	0.99	21.10	0.12	24.43	58.31	1160	0.214	0.184	1.22	0.62	17.30	0.13	21.94	59.71
rocII-4-11	426	0.061	0.144	9.16	1.58	0.44	14.53	58.19	421	0.056	0.132	9.78	0.49	0.96	0.48	15.14	57.94
rococoC10-001000	1053	0.040	0.038	1.78	7.57	0.15	11.95	60.18	1112	0.048	0.043	1.18	1.31	8.18	0.17	13.91	62.00
roll3000	948	0.089	0.094	2.54	8.46	0.11	13.69	66.11	945	0.077	0.082	2.11	0.91	7.87	0.10	13.63	66.98
satellites1-25	5040	4.185	0.830	0.81	20.39	0.03	22.85	56.72	4573	3.060	0.669	1.17	0.48	17.00	0.03	20.04	55.12
sp98ic	577	0.170	0.295	0.72	1.84	0.03	3.60	13.21	591	0.167	0.283	1.10	0.27	1.90	0.04	4.24	12.30
sp98ir	561	0.092	0.165	2.65	6.76	0.11	12.26	49.81	516	0.079	0.153	2.40	0.65	8.88	0.10	15.07	50.08
tanglegram1	362	0.620	1.713	5.67	0.16	0.32	7.34	69.78	369	0.642	1.739	5.52	0.04	0.01	0.30	6.46	71.41
tanglegram2	184	0.064	0.347	10.38	0.23	0.68	12.48	69.06	184	0.057	0.312	10.58	0.07	0.03	0.67	12.31	69.04
timtab1	178	0.002	0.011	8.59	2.54	1.08	18.19	36.27	178	0.002	0.010	8.98	1.34	0.00	1.05	17.52	33.84
triptim1	43642	112.778	2.584	3.83	9.80	0.02	14.17	61.51	44006	114.450	2.601	4.01	0.22	9.31	0.02	14.08	61.54
unitcal_7	15686	1.357	0.086	9.88	0.43	0.40	11.70	71.02	15502	1.153	0.074	13.64	0.33	0.13	0.55	15.49	65.22
vpphard	1583	2.354	1.487	1.22	10.06	0.08	12.78	57.27	1510	2.046	1.355	1.13	0.35	10.06	0.08	13.17	57.20
zib54-UUE	4492	0.456	0.101	4.97	2.89	0.12	8.88	60.24	3757	0.343	0.091	3.51	0.52	6.63	0.15	11.77	57.92
average	6263	45.290	1.025	5.52	7.55	0.34	15.48	50.75	5887	41.083	1.024	5.73	0.75	7.12	0.36	15.80	49.86
geom. mean	1229	4.014	0.797						1217	3.830	0.778						

Table 9: Running time: with and without of our permutation method (Miplib 2010 preprocessed)

instance	Suhl-Suhl update								permute + Suhl-Suhl update								
	it	time (s)	t/it (ms)	%factor				%solve	it	time (s)	t/it (ms)	%factor					%solve
				%b	%S	%[]	+					%b	%Δ	%S	%[]	+	
L1_sixm250obs	79738	3600.014	45.148	0.83	1.11	0.08	2.09	69.58	86485	3600.010	41.626	0.83	0.01	0.24	0.03	1.15	69.46
Linf_520c	32763	3600.572	109.897	87.75	0.72	0.01	88.50	6.44	32466	3603.924	111.006	89.32	0.01	0.51	0.01	89.85	5.39
buildingenergy	116212	832.834	7.167	0.46	0.07	0.05	0.61	95.49	116101	806.953	6.950	0.50	0.02	0.00	0.02	0.62	95.32
cont1	61230	3602.617	58.837	81.58	0.51	0.03	82.13	13.96	45022	3606.514	80.106	70.21	0.02	0.80	0.02	71.09	22.67
cont11	84648	3613.832	42.692	99.79	0.00	0.00	99.80	0.13	84525	3608.450	42.691	99.80	0.00	0.00	0.00	99.81	0.12
cont4	40803	3633.487	89.050	75.58	1.70	0.01	77.32	17.97	40803	3606.284	88.383	75.35	0.03	1.72	0.01	77.16	18.10
dano3mip	26355	25.354	0.962	10.66	4.24	0.05	16.01	24.11	24494	23.093	0.943	12.36	0.33	3.16	0.05	16.68	24.96
dbic1	90471	2543.061	28.109	65.82	0.01	0.20	65.84	4.98	116330	1933.765	16.623	45.33	0.01	0.01	0.12	45.38	10.02
dfl001	28257	31.834	1.127	4.50	9.45	0.03	15.06	43.94	24518	24.428	0.996	5.09	0.44	6.04	0.03	12.48	42.86
ds-big	62641	535.890	8.555	4.92	0.80	0.00	5.90	4.78	42179	352.378	8.354	3.92	0.04	0.75	0.00	4.88	4.17
fome12	464998	1467.467	3.156	33.71	1.66	0.05	35.80	34.92	97219	219.595	2.259	23.93	0.21	2.32	0.03	26.93	41.47
fome13	635128	3079.132	4.848	51.12	1.37	0.05	52.79	27.27	236861	843.338	3.560	42.09	0.14	1.39	0.04	43.96	34.12
gen4	871	5.555	6.377	78.12	6.85	0.01	85.47	11.09	866	7.109	8.209	83.55	0.12	5.02	0.00	89.04	8.31
ken-18	118347	11.785	0.100	13.23	1.87	0.80	16.66	52.96	118434	10.713	0.090	14.36	0.35	0.62	0.74	16.97	50.69
l30	11180	10.606	0.949	24.13	10.34	0.04	35.20	33.27	11664	11.777	1.010	26.19	0.54	10.32	0.04	37.88	32.27
lp22	18963	16.833	0.888	11.43	6.55	0.04	18.66	31.62	20260	18.415	0.909	13.58	0.35	6.21	0.04	20.83	30.53
mod2	44263	108.954	2.462	7.73	1.13	0.07	9.24	56.62	44185	106.075	2.401	10.02	0.15	0.95	0.08	11.45	52.45
neos	115699	2302.549	19.901	0.39	1.16	0.05	1.61	87.26	111552	2140.019	19.184	0.42	0.05	1.29	0.04	1.84	87.68
neos1	43169	1028.134	23.816	0.17	22.47	0.03	22.83	55.29	47658	1142.626	23.976	0.19	0.18	22.45	0.03	23.03	54.77
neos2	78832	2023.284	25.666	0.21	21.19	0.03	21.58	55.56	59135	1457.918	24.654	0.26	0.14	19.60	0.04	20.23	55.67
neos3	36759	3600.037	97.936	0.24	21.72	0.04	22.13	48.31	34975	3600.076	102.933	0.23	0.07	20.62	0.04	21.13	51.20
ns1644855	83255	517.098	6.211	6.20	1.45	0.05	8.04	59.11	69877	438.624	6.277	3.04	0.13	1.40	0.03	4.89	71.01
ns1687037	5913	3600.669	608.941	97.62	0.26	0.01	97.90	1.33	13357	3601.863	269.661	95.37	0.01	0.46	0.01	95.89	3.05
ns1688926	88986	3600.003	40.456	66.02	0.06	0.45	66.09	6.38	85364	3600.017	42.173	65.58	0.01	0.05	0.43	65.66	6.53
nug08-3rd	33032	735.288	22.260	58.21	10.62	0.01	69.08	19.49	29543	586.104	19.839	53.24	0.11	11.23	0.01	64.84	22.63
nug15	637187	3600.000	5.650	57.40	5.24	0.01	63.09	25.31	552483	3600.311	6.517	61.37	0.15	5.32	0.01	67.20	22.66
pds-100	265946	978.883	3.681	0.80	1.67	0.12	2.85	49.86	277868	1131.143	4.071	0.79	0.08	1.10	0.06	2.18	45.65
pds-40	122238	389.029	3.183	0.48	3.54	0.05	4.52	44.69	117622	352.594	2.998	0.52	0.12	3.17	0.03	4.14	43.46
qap12	107549	158.418	1.473	31.81	10.31	0.02	42.95	38.60	126126	230.702	1.829	42.71	0.40	8.60	0.02	52.36	32.59
qap15	408397	3600.216	8.815	71.08	4.21	0.01	75.55	16.62	550514	3600.064	6.539	61.68	0.12	4.85	0.01	67.01	22.90
rail4284	35245	3600.032	102.143	0.04	0.07	0.00	0.13	0.47	36523	3600.005	98.568	0.05	0.01	0.06	0.00	0.13	0.48
self	29770	1021.038	34.298	90.92	0.84	0.00	91.88	2.97	57127	1380.330	24.162	87.32	0.01	1.17	0.00	88.70	4.31
sgpf5y6	254527	983.087	3.862	0.93	0.94	0.13	1.97	90.74	256406	1034.897	4.036	0.92	0.08	1.24	0.05	2.35	87.11
stat96v1	17359	79.193	4.562	22.92	1.22	0.02	24.38	11.13	16534	74.303	4.494	19.28	0.13	1.32	0.02	21.00	13.03
stat96v4	42461	75.087	1.768	6.58	0.80	0.01	7.68	24.29	39574	70.925	1.792	5.69	0.10	0.67	0.01	6.76	22.91
stormG2-125	101300	18.830	0.186	4.16	0.24	0.14	4.83	85.14	101052	17.706	0.175	4.43	0.15	0.05	0.13	5.09	84.23
stormG2_1000	802907	1491.448	1.858	5.58	0.06	0.10	5.72	87.36	803485	1471.501	1.831	5.51	0.03	0.01	0.09	5.64	87.11
stp3d	99833	565.454	5.664	2.00	1.53	0.05	3.67	61.83	102296	597.167	5.838	1.99	0.08	1.33	0.04	3.55	62.62
watson_2	478937	3143.706	6.564	2.77	2.17	0.20	5.12	84.26	478328	2892.521	6.047	3.67	0.07	1.98	0.06	5.86	81.38
world	238315	935.109	3.924	12.13	0.92	0.07	13.38	46.68	49651	129.860	2.615	9.81	0.13	0.90	0.08	11.18	50.34
average	151112	1619.160	36.079	29.75	4.03	0.08	34.10	38.29	128987	1478.353	27.408	28.51	0.13	3.72	0.06	32.67	38.91
geom. mean	72568	630.194	14.444						66512	540.739	13.535						

Table 10: Running time: with and without of our permutation method (Mittelmann testset)