Mathematical Programming Computation
https://doi.org/10.1007/s12532-020-00188-1

FULL LENGTH PAPER

A triangulation and fill-reducing initialization procedure for the simplex algorithm

Nikolaos Ploskas[1] · Nikolaos V. Sahinidis[2] · Nikolaos Samaras[3]

Received: 8 February 2017 / Accepted: 3 June 2020
© Springer-Verlag GmbH Germany, part of Springer Nature and Mathematical Optimization Society 2020

Abstract

The computation of an initial basis is of great importance for simplex algorithms since it determines to a large extent the number of iterations and the computational effort needed to solve linear programs. We propose three algorithms that aim to construct an initial basis that is sparse and will reduce the fill-in and computational effort during LU factorization and updates that are utilized in modern simplex implementations. The algorithms rely on triangulation and fill-reducing ordering techniques that are invoked *prior* to LU factorization. We compare the performance of the CPLEX 12.6.1 primal and dual simplex algorithms using the proposed starting bases against CPLEX using its default crash procedure over a set of 95 large benchmarks (NETLIB, Kennington, Mészáros, Mittelmann). The best proposed algorithm utilizes METIS (Karypis and Kumar in SIAM J Sci Comput 20:359–392, 1998), produces remarkably sparse starting bases, and results in 5% reduction of the geometric mean of the execution time of CPLEX's primal simplex algorithm. Although the proposed algorithm improves CPLEX's primal simplex algorithm across all problem types studied in this paper, it performs better on hard problems, i.e., the instances for which the CPLEX default requires over 1000 s. For these problems, the proposed algorithm results in 37% reduction of the geometric mean of the execution time of CPLEX's primal simplex algorithm. The proposed algorithm also reduces the execution time of CPLEX's dual simplex on hard instances by 10%. For the instances that are most difficult for CPLEX, and for which CPLEX experiences numerical difficulties as it approaches the optimal solution, the best proposed algorithm speeds up CPLEX by more than 10 times. Finally, the proposed algorithms lead to a natural way to parallelize CPLEX with speedups over CPLEX's dual simplex of 1.2 and 1.3 on two and four cores, respectively.

Keywords Linear programming · Revised simplex algorithm · Initial basis · Crash procedure

---

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s12532-020-00188-1) contains supplementary material, which is available to authorized users.

Extended author information available on the last page of the article

Published online: 26 June 2020

Mathematics Subject Classification 90C05 · 90C06 · 65K05

1 Introduction

Since the introduction of the simplex algorithm in 1947 [9,10], Linear Programming (LP) has been widely used in many application areas in science and engineering and led to the genesis of the mathematical programming community [31]. Since that time, a variety of algorithmic and computational techniques have been developed to improve the computational performance of the simplex algorithm:

- Presolve methods that reduce the problem size [24,36,47] (for a review, see [3]).
- Scaling techniques that improve the numerical behavior of the simplex algorithm and reduce the number of iterations required to solve LPs [8,17,45] (for a review, see [43]).
- Pivoting rules that reduce the number of simplex iterations required to solve LPs [19,26,44] (for a review, see [42]).
- Basis factorization and update methods that improve the numerical behavior of the simplex algorithm and reduce its execution times [20,22,33] (for reviews, see [15,16]).

The simplex algorithm starts with a feasible basis and uses pivot operations in order to preserve feasibility of the basis and guarantee monotonicity of the objective value. In some very simple cases, a basic feasible solution may be available.

The quality of the initial basis greatly affects the execution time, the number of iterations, and the required storage of the algorithm's data structures [5,23,34,35,40]. The aim of the crash procedures is to find an initial basis that: (i) is close to optimal, (ii) is sparse, (iii) will reduce the subsequent fill-ins of the LU factorization, (iv) will reduce the execution time per iteration, and (v) will reduce the number of iterations. Crash procedures may sometimes increase the number of iterations but they may also achieve a decrease in the time per iteration and the overall execution time. Most crash procedures use triangulation and sparsification concepts. Considering that the initial basis will be factorized using LU decomposition, most crash procedures form a nearly-triangular and sparse basis that is likely to limit the number of subsequent fill-ins.

Considerable attention has been given to the initialization of the simplex algorithm since its conception. Most linear programming textbooks [4,7,34] present only simple initialization procedures, such as the all-artificial and the slack-artificial basis. Twelve different initialization techniques have been developed for general LPs; six additional techniques have been developed for LPs with special structure. Most notably, advanced crash procedures for initializing the simplex algorithm have been proposed in [5,6,23,35,37]. Initialization procedures that can be applied in special cases or in modified simplex-type algorithms have been presented in [1,25,28,32,38,39]. All these crash procedures will be reviewed in detail in Sect. 2.

This paper proposes new methods for initializing the simplex algorithm. The overall goal of these methods is to exploit the concepts of triangulation and sparsification in order to create a nearly-triangular and sparse basis that will limit the number of

fill-ins of the LU factors of the bases generated by the simplex algorithm. The triangulation step is achieved via permutation of column singletons of the LP problem matrix to identify a maximal submatrix that includes columns of the identity matrix. The sparsification step relies on fill-reducing strategies that have been devised to minimize the maximum potential fill-in in LU factorization procedures. These fill-reducing strategies have been designed for factorizing symmetric matrices in the context of LU factorization. However, for crash procedures based on these strategies, the impact on the performance of modern simplex codes is unknown. Given the obvious relative advantages and disadvantages of starting points that are sparse but far from optimal versus starting points that are less sparse but nearly-optimal, we propose to investigate the impact of these strategies computationally. We thus apply them to the nonsingleton columns of the constraint matrix for the purpose of supplementing column singletons with additional columns that are likely to lead to minimal fill-in in the subsequent LU factorization and update procedures during simplex iterations. In general, finding a permutation matrix that minimizes fill-in is NP-complete [46]. For this reason, heuristics are used to find good orderings. In this paper, we experiment with three different fill-reducing ordering methods: (i) COLAMD [13], (ii) AMD [2], and (iii) METIS [30]. Even though these techniques have not been considered in the numerical linear algebra of the simplex algorithm, we will demonstrate that they can provide starting bases that, in comparison to existing implementations, are sparser and reduce the fill-in and computational effort during LU factorization and updates for many LPs.

The remainder of this paper is organized as follows. In Sect. 2, we review procedures for finding an initial basis. Section 3 presents the proposed methods. Section 4 presents results from an extensive computational study that compares the performance of the proposed methods against the default CPLEX crash procedure. Conclusions are provided in Sect. 5.

2 Review of crash procedures

The aim of a crash procedure is to find an initial basic solution. The starting basis may be feasible or infeasible. In case the basis is feasible ($l_B \leq x_B \leq u_B$, where B is the set of the basic variables, l and u are the lower and upper bounds of the variables) simplex algorithms can use it as a starting solution and proceed to find a solution of the problem. On the other hand, if the initial basis is not feasible, different methods can be used to find a basic feasible solution. Three methods are primarily used: (i) the two-phase method, (ii) the big-M method, and (iii) the single artificial variable method. Modern implementations of the simplex algorithm use the two-phase method.

Let's assume that the LP is in the so called computational form:

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & l \leq x \leq u \end{aligned}$$

where $c, l, x, u \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times n}$, and T denotes transposition. Assume that A has full row rank and contains (implicitly) an identity matrix.

The two-phase method adds an artificial variable to each constraint and solves an auxiliary LP in Phase I:

$$\begin{aligned} \min \quad & e^T y \\ \text{s.t.} \quad & Ax + I_m y = b \\ & x, y \geq 0 \end{aligned}$$

where $e \in \mathbb{R}^m$ is a vector of ones and I_m is an identity matrix of size $m \times m$. The auxiliary LP is solved using the simplex algorithm. If $y \neq 0$ at optimality, then the original LP is infeasible. If $y = 0$, then there are two possibilities:

- $y = 0$ and no auxiliary variable is in the basis: in this case, we have identified a basic feasible solution $x = [x_B, x_N]^T$, where B is the set of the basic variables and N is the set of the nonbasic variables. The nonzero elements in x form x_B; the remaining form x_N. We can solve the original LP starting with this basic feasible solution after eliminating the artificial variables and the corresponding columns from the problem.
- $y = 0$ and at least one auxiliary variable is still in the basis: in this case, we have identified a degenerate solution to the auxiliary problem. We remove the artificial variables from the basis. If the lth variable is an artificial variable, examine the lth element of the columns $A_B^{-1} A_{.j}$, $j = 1, \ldots, n$. If the lth element of the jth column is nonzero, then apply a change of basis with the lth entry serving as the pivot element. The lth basic variable exits the basis and variable x_j enters the basis.

If the initial basis is not feasible, LP solvers search for a feasible point during Phase I. Hence, a crash procedure that produces feasible starting bases avoids Phase I and may lead to fewer simplex iterations. Nonetheless, the problem of finding a feasible point has the same complexity bound as the linear programming problem [41].

The simplest initial basis is the all-artificial basis or all-logical basis, presented in most linear programming textbooks [4,7,34]. Artificial variables are added to all constraints and the initial basis consists of the artificial variables. The all-artificial basis is extremely simple and has three distinct advantages [34]: (i) its creation is instantaneous, (ii) the LU decomposition of the starting basis (I) is available, and (iii) the first iterations are very fast as the operations utilize a very sparse LU factorization. Another simple initial basis is the slack-artificial basis [5]. Initially, we add slack and surplus variables to all inequality constraints. Then, we add artificial variables to equality constraints and inequality constraints of the type $\geq$. The initial basis is formed by the slack variables added in inequality constraints of type $\leq$ and the artificial variables. The slack-artificial basis is better than the all-artificial basis since it adds fewer artificial variables and solves a smaller LP in Phase I. The techniques discussed in this paragraph are known to lead to substantially larger numbers of iterations than other initialization techniques.

A variant of the slack-artificial initial basis is the feasible slack basis [5]. In this method, we add slack and surplus variables to all inequality constraints. Then, we add artificial variables to all constraints and form an initial basis consisting of only the artificial variables. Next, available slacks that are initially nonnegative replace the artificial variables in the basis. Bixby [5] also proposed an approach to create a sparse

and well-behaved basis with as few artificial variables as possible. The generated basis includes all slack variables. The remainder of the structural variables are assigned a preference order of inclusion in the basis; this preference order aims to place the variables with the most freedom at the start of the list using the objective function to break ties. Then, a heuristic procedure selects the variables that will be included in the basis aiming to form a nearly triangular basis. Bixby's computational results suggested that his basis can greatly reduce the number of iterations, especially for easy problems, but it is generally less effective for harder problems.

Carstens [6] classifies crash procedures into two classes: *GAIN switch on* and *GAIN switch off*. In the GAIN switch off case, the objective function is ignored and the starting basis is chosen based on sparsity grounds alone. Carstens assumes that a starting set of basic variables is given as input to the crash procedure. It may consist entirely of artificial variables in case there is no information about selecting basic variables. At each iteration of these crash procedures, a pivot element a_{ij} is selected to replace column i of B with column j of A. If column j has c_j nonzeros and row i has r_i nonzeros, Carstens discusses three different ways to select a pivot element:

- Order the nonbasic columns in order of increasing c_j and choose the pivot element a_{ij} to be a nonzero that minimizes r_i.
- Order the rows in order of increasing r_i and choose the pivot element a_{ij} to be a nonzero that minimizes c_j for j nonbasic.
- Consider the nonzeros in increasing order of the count $(r_i - 1)\left(c_j - 1\right)$ (Markowitz criterion for reinversion [33]).

In the GAIN switch on case, a basis change is made only if it leads to an improvement in the objective function. Carstens recommends the use of the GAIN switch off when the starting basis is totally or mostly artificial and the GAIN switch on when the starting basis includes few artificial variables. Reid developed an algorithm, presented by Gould and Reid [23], that forms an upper triangular basis. In comparison to Carsten's GAIN switch off algorithm, a column that is chosen late in Reid's algorithm is required to have a nonzero in at least one row that has not yet been pivotal. Gould and Reid [23] proposed a tearing crash procedure that aims to find an initial basis that is as feasible as possible and can be calculated with a reasonable computational effort. The approach relies on the P5 algorithm of Erisman et al. [18] and solves a series of small LPs, the solution of which forms a basis for the initial LP. Maros and Mitra [35] proposed four crash procedures: (i) CRASH(LTSF): a lower triangular symbolic crash procedure designed for feasibility, (ii) CRASH(ADG): an anti-degeneracy crash procedure that deals with LPs where a starting basis may lead to a primal degenerate solution, (iii) CRASH(ART): an artificial removal technique used after CRASH(LTSF), and (iv) CRASH(SOR): an iterative crash procedure based on Kaczmarz's SOR algorithm [29]. MINOS [37] contains a crash procedure where a pivot a_{ij} is selected if its row contains zeros in all the columns that have so far been chosen as basic or if its column contains zeros in all the rows that have been pivotal.

Al-Najjar and Malakooti [1] use a Phase I method that moves through the interior of the feasible region to obtain an initial basic feasible solution. Gülpinar et al. [25] proposed a method to construct an initial basis for LPs with embedded pure network structures. Junior and Lins [28] estimate an optimal (or near-optimal) basis by finding

constraints which intersect the gradient plane at minimal angles. Luh and Tsaih [32] developed a search direction that combines the gradient direction and an internal pointing direction with respect to the polyhedron forming the feasible region. Nabli [38] proposed a method for constructing an initial feasible solution from an infeasible one. This method operates without artificial variables and without any perturbation in the objective function. Feasibility is obtained via a modification of the structure of the simplex algorithm in the choice of the entering and leaving variables. Nabli and Chahdoura [39] presented a crash procedure that does not involve any artificial variables and can also detect redundant constraints and infeasibility.

The majority of the state-of-the-art crash procedures focus on finding an initial basis that is as close to optimality as possible without aiming to create a sparse initial basis that will limit the number of fill-ins of the LU factors of the bases generated by the simplex algorithm. In this paper, we investigate whether it may be better–at least in certain cases–to rely on a crash procedure that aims to choose an initial basis in a way that will be very sparse and nearly triangular. Even though it is counter-intuitive that it would be advantageous to use a crash procedure that ignores the objective function, a sparse and near triangular initial basis is more likely to minimize the subsequent fill-ins during the LU factorization of the simplex bases. All state-of-the-art LP solvers apply such techniques to factorize bases in the course of the algorithm. Our proposal is to utilize these techniques *also* for the construction of the initial basis and investigate the computational impact of this approach on primal and dual simplex algorithms.

3 The proposed algorithms

In this section, we present three algorithms to construct an initial basis for the simplex algorithm. All proposed algorithms ignore the objective function and the bounds of the variables and choose the initial basis in a way that it will be very sparse and nearly triangular. The motivation of these algorithms is to quickly find a starting basis that is likely to minimize subsequent fill-ins during the LU factorization of the simplex bases. The first step in all algorithms is to identify a maximal submatrix of A that is a diagonal. In particular, if a column singleton a_{ij} exists, its column j is permuted to the left and its row i is permuted to the top. Column j and row i are removed from A. Such singleton columns must be present in the original constraint matrix A, not just in the matrix remaining once pivoted rows and columns have been removed. The process repeats until no more singletons exist, leading to

$$\begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix}$$

where A_{11} is a diagonal matrix whose diagonal entries are greater than the smallest acceptable pivot value $\tau > 0$. The computational effort of this procedure depends on the kinds of data structures used. In one implementation, the time to find all singletons and permute them to the top left corner of the constraint matrix is reported to be $O(n + |A_{11}| + |A_{12}|)$ [13], where $|A|$ denotes the number of nonzeros of matrix A.

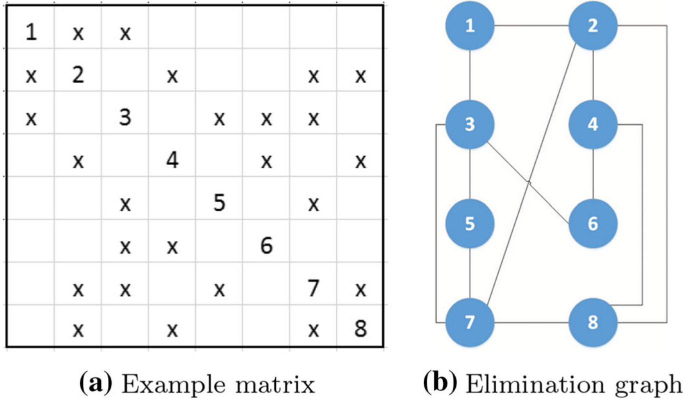


Fig. 1 Example matrix and its elimination graph

If the sum of the number of $\leq$ type of constraints and the singleton columns in the original LP problem is m, initialization stops here with a basis consisting of all slack variables and/or variables with singleton columns.

Once singletons are removed, the remaining matrix A_{22} is ordered with a fill-reducing ordering method. The goal of this procedure is to find a column permutation of A_{22} so that subsequent factorization results in the least possible fill-in in A_{22}. The output of this procedure is a column permutation vector. We use this column permutation vector to select the initial basis for the simplex algorithm. The initial basis will be formed by the s singleton columns ($0 \leq s \leq n$, if $s > m$ we select the first m singletons as the initial basis) and the first $m - s$ columns from the column permutation vector.

The column preordering is based solely on the nonzero pattern of A_{22}. Some methods order matrix A without forming $A^T A$, while others form the explicit pattern of $A^T A$. The nonzero pattern of the symmetric $n_2 \times n_2$ matrix $A_{22}^T A_{22}$ (where n_2 is the number of columns of matrix A_{22}, $n_2 \leq n$) can be represented by a graph $G^0 = \left(V^0, E^0\right)$, where $V^0 = \{1, \ldots, n_2\}$ are the nodes and E^0 are the edges of the graph. An edge $(i, j) \in E^0$ if and only if $a_{ij} \neq 0$ and $i \neq j$. Since the matrix is symmetric, G^0 is undirected. Figure 1 illustrates an example matrix and its elimination graph G^0.

If A_{22} contains a dense (or nearly dense) row or column, the Markowitz criterion will not chose this row or column until the final stages of the elimination, thus limiting fill-in, which is consistent with our intent to produce a sparse starting basis.

As already mentioned, because the problem of obtaining an ordering with minimum fill-in is NP-complete, heuristics are applied for choosing the pivot columns in LU factorization. In each factorization step, COLMMD [21] selects as pivot the column that minimizes a loose upper bound on the external row degree. AMD [2] is based on a bound on the external row degree that is tighter than the COLMMD bound. The Markowitz rule [33] selects as pivot the element a_{ij} that minimizes the product of the degrees of row i and column j. COLAMD [13] uses an initial COLMMD metric and an AMD metric during the elimination phase. METIS [30] finds a fill-reducing

ordering for a symmetric sparse matrix via recursive nested dissection. Amestoy et al. [2] performed a computational study in the context of minimum degree orderings for sparse Cholesky factorization and found that AMD is superior to the COLMMD approximation. In addition, Davis et al. [13] compared the performance of COLAMD, COLMMD, and AMD. Computational results showed that, for square nonsymmetric matrices, COLAMD is much faster and provides better orderings than COLMMD. For rectangular matrices, COLAMD is faster than COLMMD and AMD and finds orderings of comparable quality. Hence, we selected COLAMD, AMD, and METIS to create variants of our method. COLAMD orders matrix A without forming $A^T A$, while AMD and METIS need to form the explicit pattern of $A^T A$. The asymptotic run times of these ordering methods have no tight known bounds in terms of quantities that can be readily calculated beforehand [11]. However, experimental results presented in [12] showed that, in most cases, COLAMD and AMD take time roughly proportional to the number of nonzeros in A and $A^T A$, respectively.

All algorithms select the same singleton columns to include in the initial basis. Their only difference is the ordering method. Therefore, the three variants of the proposed method are:

- Algorithm 1 applies COLAMD.
- Algorithm 2 applies AMD.
- Algorithm 3 applies METIS.

We also experimented with using the Markowitz [33] criterion to select the basis but this approach leads to more simplex iterations. These results are consistent with the results of Davis et al. [12], who also considered the Markowitz criterion prior to the LU factorization in order to permute a matrix and reduce the worst-case fill-in. They report that the Markowitz criterion gave much worse orderings than COLAMD. In our case, these worse orderings resulted in more simplex iterations.

The input to all three algorithms is the constraint matrix A and the output is the basic list B. The basic steps of the aforementioned algorithms can be described as follows:

Step 1. Set $C = \varnothing$, $R = \varnothing$ and $Q = \varnothing$.

Step 2. Find the singletons in the constraint matrix A. A singleton is a column j with a single nonzero a_{ij} whose magnitude is larger than a given threshold τ. We set $\tau = 20\,(m+n)\,\epsilon \max_j \left\| A_{*j} \right\|_2$, where ϵ is the machine roundoff and $\max_j \left\| A_{*j} \right\|_2$ is the largest 2-norm of any column of A. If a singleton a_{ij} exists and $i \notin R$, add column j to the set C and row i to the set R. If $|C| = m$, go to Step 4; else, repeat this step until there are no more singletons.

Step 3. Apply COLAMD (for Algorithm 1), AMD (for Algorithm 2), or METIS (for Algorithm 3) to submatrix A_{22} (A_{22} is a submatrix of A by deleting rows in A that are in set C and columns that are in set R). The resulting column permutation vector is stored in set Q.

Step 4. The initial basic list is B formulated from the variables in set C and the first $m - |C|$ variables in set Q.

Note that we can create additional variants for each of the proposed methods if we permute the rows in R and columns in C of the constraint matrix A to the top left

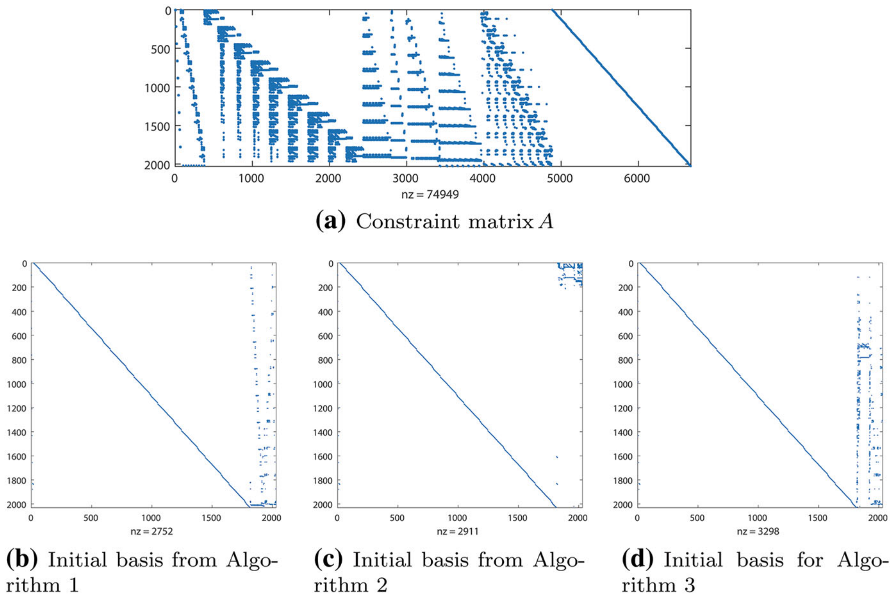


Fig. 2 Sparsity pattern of the constraint matrix A and the initial basis using Algorithms 1–3 for problem pilot87 of the NETLIB set

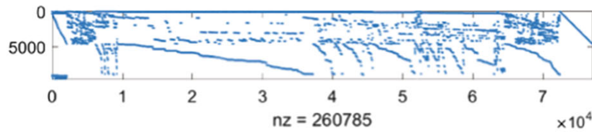
corner. A preliminary computational study revealed that these permutations result in more iterations and slower execution times of the simplex algorithm. Hence, these variants will not be discussed further.

The proposed algorithms do not guarantee that the initial matrix will be nonsingular since the ordering methods that are used (AMD, COLAMD, METIS) do not guarantee that the matrices generated by their orderings will be nonsingular. In fact, we were able to generate some trivial instances for which the ordering methods generate an ordering that does make our algorithm produce a singular initial matrix. However, the proposed method did not generate a singular initial matrix for any of the benchmark problems we experimented with (from NETLIB, Kennington, Mészáros, Mittelmann benchmark libraries).

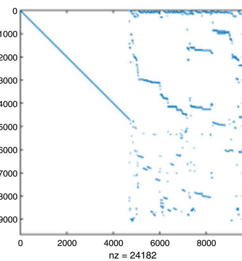
Figures 2 and 3 present the sparsity pattern of the constraint matrix A and the initial basis using Algorithms 1–3 for problems pilot87 and qap15 from NETLIB, respectively. All algorithms form nearly-triangular initial bases.

4 Computational study

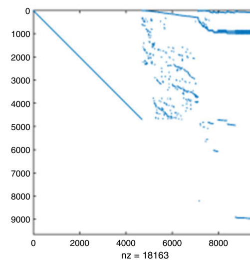
The aim of this computational study is to investigate the performance of the simplex algorithm in conjunction with the proposed crash procedures. We give the initial bases generated by all three algorithms as input to the CPLEX solver and compare their performance against the CPLEX default crash procedure. We do this using both the primal and the dual simplex algorithm.



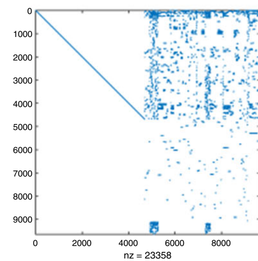
(a) Constraint matrix A



(b) Initial basis from Algorithm 1



(c) Initial basis from Algorithm 2



(d) Initial basis for Algorithm 3

Fig. 3 Sparsity pattern of the constraint matrix A and the initial basis using Algorithms 1–3 for problem qap15 of the NETLIB set

All computations were performed on an Intel Xeon CPU E5-2660 v3 with 128 GB of main memory, a clock of 2.6 GHz, an L1 code cache of 32 KB per core, an L1 data cache of 32 KB per core, an L2 cache of 256 KB per core, and an L3 cache of 24 MB, running under Centos 7 64-bit. We considered a set of 150 medium-sized and large benchmark problems (NETLIB, Kennington, Mészáros, Mittelmann) in preliminary runs. Then, we eliminated the trivial problems, i.e., instances solved in less than 1 s with all the algorithms considered in this paper when CPLEX presolve is disabled ("preprocessing.presolve" option is set to 0). The final set of instances that we used in this computational study includes 95 benchmark problems. On average, 6% of the variables in the constraint matrix are singletons while 10% of the variables in the initial basis are singletons. Table S1 in the Online Supplement presents the number of constraints, variables, and nonzeros for each of the benchmark problems. We used CPLEX to presolve all instances and exported the MPS files. We then generated the initial bases for each presolved problem using the three algorithms and stored them in BAS files (MPS basis files, known as BAS files, that contain the information needed to define an initial basis). We gave the generated BAS files as input to CPLEX and compared the performance of the solver against that of the CPLEX default crash procedures. We did this comparison for both the primal and the dual simplex algorithm. We used default values for all algorithmic options of CPLEX. An execution time limit of 15,000 s was imposed on all runs.

In the tables and figures below, the following abbreviations are used: (i) Time: CPU time to solve a problem with CPLEX, and (ii) Tit: total iterations. The time to construct an initial basis with the proposed algorithms is negligible in comparison to the total time needed to solve the instances. Algorithm 1 (based on COLAMD) is faster than Algorithms 2 (based on AMD) and 3 (based on METIS).

Table 1 Shifted geometric times and iterations for the primal simplex algorithm using shifted geometric mean

Algorithm	Test set	Time	Tit
CPLEX using Algorithm 1	All problems[a]	56	41,921
	> 1000 s[b]	3334	308,677
CPLEX using Algorithm 2	All problems	58	41,639
	> 1000 s	3626	348,100
CPLEX using Algorithm 3	All problems	55	40,485
	> 1000 s	2885	300,005
CPLEX using default crash procedure	All problems	58	43,156
	> 1000 s	4606	348,349

[a] 95 problems in total
[b] 13 problems for which CPLEX with default crash needs more than 1000 s to solve

Table 1 presents the average value (shifted geometric mean over the entire collection of test problems) of Time and Tit with four different initialization algorithms followed by the application of the primal CPLEX routine to the presolved problems. For the nonnegative numbers $a_1, \ldots, a_k \in \mathbb{R}_+$ and a shift $s \in \mathbb{R}_+$, the average is defined by

$$\gamma_s\left(a_1, \ldots, a_k\right) = \left(\prod_{i=1}^{k}\left(a_i + s\right)\right)^{\frac{1}{k}} - s$$

We use a shift of 10 for the execution time and 1000 for the number of iterations in order to decrease the influence of the easy instances in the mean values.

Tables S2–S5 in the Online Supplement present the detailed results for each problem and algorithm combination. As seen in Tables 1 and S2–S5, Algorithm 3, based on METIS, performs better than all the other proposed methods on average. All the proposed methods require less CPU time and fewer iterations than the default CPLEX crash procedure. Primal CPLEX using Algorithm 3 results in 5% reduction of the geometric mean of the execution time of CPLEX's primal simplex algorithm. Moreover, the proposed methods are significantly faster on instances for which the CPLEX default requires over 1000 s (13 problems). For these problems, primal CPLEX using Algorithm 3 is 37% faster than primal CPLEX using its default crash procedure.

Figures 4 and 5 present performance profiles [14] based on the execution time and the number of iterations, respectively, of the primal simplex algorithm using the three proposed algorithms and the default crash procedure. Performance profiles are displayed in logarithmic scale with base 2. Algorithm 3, based on METIS, performs better than the other proposed methods and the default crash procedure. In particular, Algorithm 3 is better than the other methods in the interval [1.1, 7]. Moreover, Algorithm 3 is faster than the CPLEX default crash procedure on 64 out of 95 problems and appears dominant in the performance profile. Algorithm 3 performs 4% fewer Phase I iterations, 2% fewer Phase II iterations, and 6% fewer total iterations than the CPLEX crash procedure. The proposed algorithm performs fewer Phase I iterations

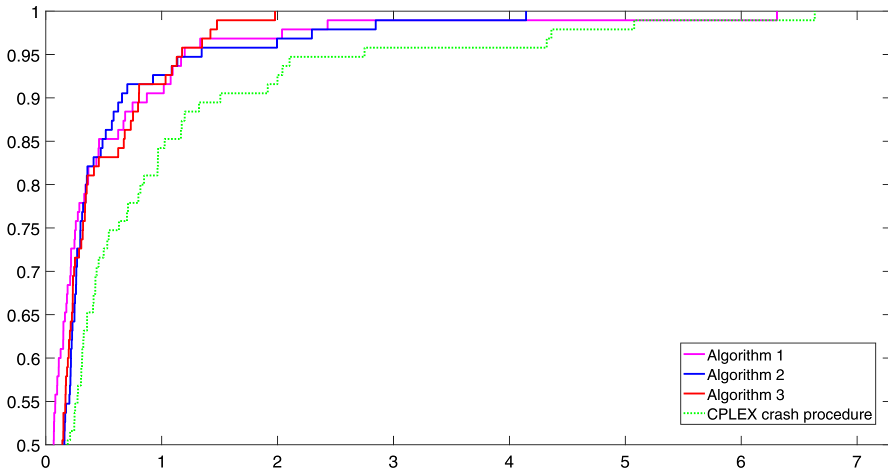


Fig. 4 Performance profiles comparing the three algorithms and default crash procedure based on the execution time for the primal simplex

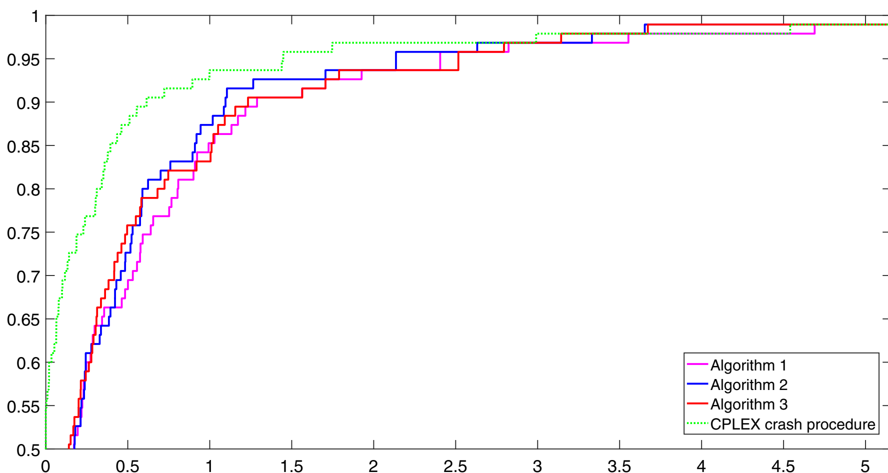


Fig. 5 Performance profiles comparing the three algorithms and default crash procedure based on the number of iterations for the primal simplex algorithm

on 51 instances, fewer Phase II iterations on 48 instances, and fewer total iterations on 47 instances. Algorithm 3 finds a better starting solution (closer either to feasibility or optimality) than the CPLEX crash procedure on 61 problems. Additionally, Algorithm 3 finds the optimal solution on one problem, a basic feasible solution on six problems and a nearly basic feasible solution (the percentage of Phase I iterations to total iterations is less than 10%) on 18 problems, while the CPLEX crash procedure finds a basic feasible solution on one problem and a nearly basic feasible solution on 27 problems. In addition, Algorithm 3 constructs an initial basis that is, on average, four times sparser than that of the CPLEX crash procedure.

Table 2 Shifted geometric means of times and iterations for the dual simplex algorithm

Algorithm	Test set	Time	Tit
CPLEX using Algorithm 1	All problems[a]	51	27,854
	> 1000 s[b]	8674	333,956
CPLEX using Algorithm 2	All problems	50	26,901
	> 1000 s	8782	327,671
CPLEX using Algorithm 3	All problems	50	27,281
	> 1000 s	7074	306,132
CPLEX using default crash procedure	All problems	48	24,345
	> 1000 s	7844	291,366

[a]95 problems in total
[b]8 problems for which CPLEX with default crash needs more than 1000 s to solve

Although the performance of Algorithm 3 is consistent on both easy and hard instances, it results in significant reductions when solving hard instances. The performance of CPLEX with its default crash procedure deteriorates for large and hard problems. More specifically, there are some problems, e.g., neos2, ns1687037, nug08-3rd, and nug20, where CPLEX experiences numerical difficulties as it approaches the optimal solution. These difficulties caused CPLEX to change the value of the Markowitz tolerance and resort to new Phase I iterations in order to restore feasibility. CPLEX may start again from an infeasible solution more than once during the solution of a problem, e.g., four and six times for the ns1687037 and nug20 instances, respectively. CPLEX also experienced numerical issues when starting from a solution generated by one of the proposed algorithms. In all such cases, however, CPLEX needed only a few iterations to restore a feasible solution. Therefore, the proposed methods seem to have the ability to avoid numerical issues encountered by the starting points obtained through the current default initialization algorithms in CPLEX.

Table 2 presents a summary of the results for the dual simplex algorithm. Detailed results with all problem and algorithm combinations are provided in Tables S6–S9 in the Online Supplement. In this case, CPLEX's dual simplex algorithm using the default crash procedure is 5% faster than CPLEX's dual simplex algorithm using Algorithm 3. However, Algorithm 3 is significantly better on instances for which the CPLEX default requires over 1000 s (8 problems). For these instances, dual CPLEX using Algorithm 3 is 10% faster than dual CPLEX using its default crash procedure. In addition, Algorithm 3 is performing better than Algorithms 1 and 2.

Figures 6 and 7 present performance profiles based on the execution time and the number of iterations, respectively, of the dual simplex algorithm using the three proposed algorithms and the default crash procedure. CPLEX default crash procedure has the highest probability of being the fastest solver for values of τ in the interval [0.5, 7]. CPLEX using its default crash procedure is 5% faster than CPLEX using Algorithm 3. The reduction to the execution time that the proposed algorithms offer is more pronounced on hard instances. For these problems, CPLEX using Algorithm 3 is 10% faster than the default CPLEX crash procedure.

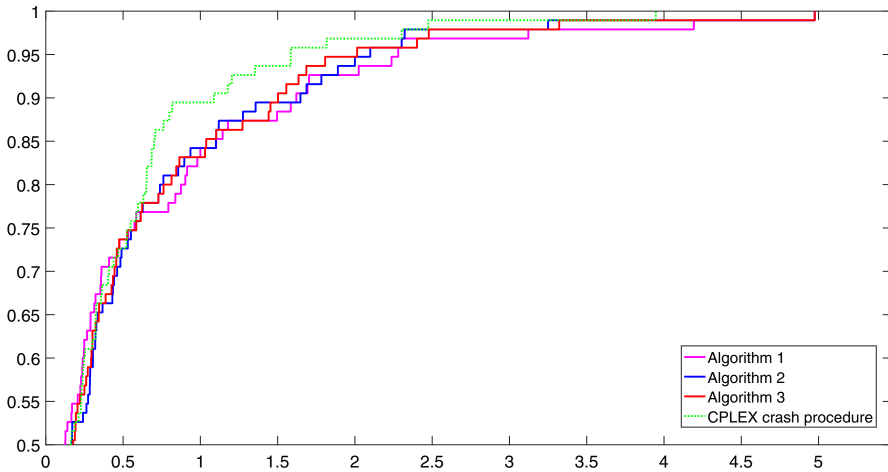


Fig. 6 Performance profiles comparing the three algorithms and default crash procedure based on the execution time for the dual simplex algorithm

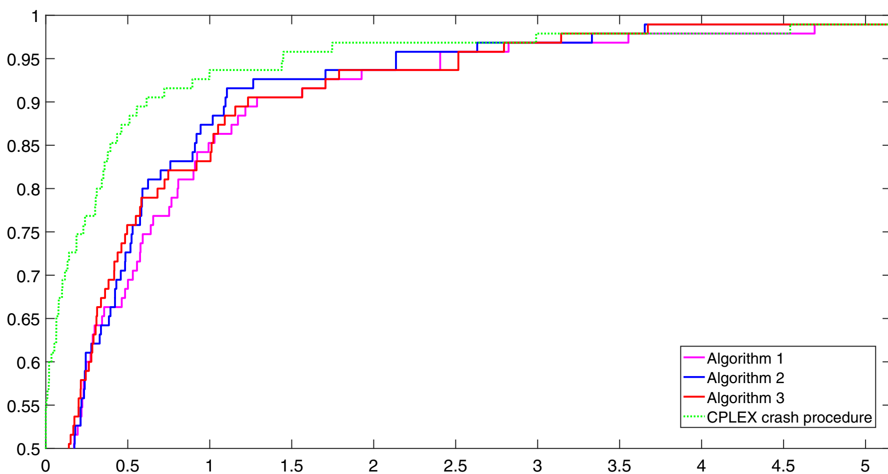


Fig. 7 Performance profiles comparing the three algorithms and default crash procedure based on the number of iterations for the dual simplex algorithm

Using dual simplex, the CPLEX crash procedure is faster than Algorithm 3 on 52 out of 95 instances. CPLEX crash procedure performs 52% more Phase I iterations, 1% more Phase II iterations, and 11% more total iterations in comparison to Algorithm 3. The CPLEX crash procedure also finds a feasible solution on the majority of the instances. Algorithm 3 performs fewer Phase I iterations on 28 instances, fewer Phase II iterations on 39 instances, and fewer total iterations on 33 instances. For the hard instances alone, Algorithm 3 results in great reductions compared to the CPLEX dual simplex algorithm. Similar to the primal simplex algorithm, the performance of the

Table 3 Average performance of the best proposed method and the best CPLEX crash procedure

Algorithm	Test set	Time	Tit	Density (%)
Primal CPLEX using Algorithm 3	All problems[a]	838	148,835	0.11
	> 1000 s[b]	7378	597,926	0.01
Dual CPLEX using default crash procedure	All problems	903	90.884	0.50
	> 1000 s	9825	442,275	0.01
Speedup of the best proposed method over the best CPLEX crash procedure	All problems	7%	– 39%	73
	> 1000 s	25%	– 26%	0

[a]95 problems in total
[b]8 problems for which CPLEX with default crash needs more than 1000 s to solve

Table 4 Shifted geometric means of wall-clock times from runs on multiple cores

Algorithm	Time
Dual CPLEX using default crash procedure	48
Best of primal and dual CPLEX using Algorithm 3	40
Best of primal and dual CPLEX using default crash procedure or Algorithm 3	37

CPLEX dual simplex algorithm with the CPLEX default crash procedure deteriorates for large and hard problems.

7

Table 3 presents the average performance of the primal simplex algorithm using Algorithm 3 compared to the performance of the dual simplex algorithm using CPLEX's default crash procedure. Performance is measured in terms of execution time, number of iterations, and density of the generated basis. CPLEX's primal simplex algorithm initialized with Algorithm 3 is 7% faster than the default CPLEX algorithm. This CPU time reduction comes along with a 73% reduction in the density of the generated initial bases. For the instances for which the dual simplex algorithm with CPLEX's default crash procedure needs more than 1000 s to solve, CPLEX's primal simplex algorithm using Algorithm 3 is 25% faster than the default CPLEX algorithm. Even though CPLEX with Algorithm 3 performs more iterations than the default CPLEX algorithm, Algorithm 3 spends significantly less time per iteration than CPLEX with the default crash procedure, for both primal and dual simplex.

The above computational results suggest there are many problems for which the proposed algorithms outperform the CPLEX default initialization scheme, while the latter is still useful, especially for easier problems. This observation suggests an opportunity to combine all these algorithms in a speculative parallelization approach on computing equipment with a small number of cores. CPLEX has no parallel simplex facility. Hence, we will compute parallelization speedups with respect to running the dual CPLEX algorithm on a single core. Table 4 presents the shifted geometric means

of the execution times when using multiple cores, each core running CPLEX with a different variant of the crash procedure in a task-dependent fashion. The default dual CPLEX using the default CPLEX crash procedure (running on one core) needs 48 s on average to solve the problems in our testset. Running the primal and dual CPLEX using Algorithm 3 on two cores and taking the best performance of each variant results in a mean speedup of 1.2 over CPLEX's dual simplex algorithm. The execution of the primal and dual CPLEX using the default crash procedure and Algorithm 3 (running on four cores) results in a mean speedup of 1.3 over CPLEX's dual simplex algorithm. These speedups are comparable to those of state-of-the-art parallel simplex solvers for a similar number of cores [27].

5 Conclusions

We presented three algorithms that construct a nearly-triangular and sparse initial basis for the simplex algorithm. The initial basis is artificial-free and includes as many structural variables as possible. The aim of the proposed methods is to reduce the subsequent fill-ins of the LU factorization, the number of iterations, and the computational effort at each iteration. We experimented with various ordering methods in order to create a sparse nearly-triangular initial basis for the simplex algorithm. Using a collection of 95 benchmark LPs, we found that the best way to speed up the primal and dual simplex algorithms for CPLEX is to utilize Algorithm 3, which forms a starting basis using all available column singletons plus the columns obtained from the application of METIS to the remainder of the LP matrix.

Algorithm 3 results in 5% average reduction of the execution time of CPLEX's primal simplex algorithm. Although the proposed algorithm reduces CPLEX's execution time on the majority of instances, it is significantly faster than the CPLEX default crash procedure on hard instances. For the hard instances (instances that CPLEX needs more than 1000 s to solve), Algorithm 3 results in 37% average reduction of the execution time of CPLEX's primal simplex algorithm. CPLEX's dual simplex algorithm with its default crash procedure is 5% faster than CPLEX's dual simplex algorithm using Algorithm 3. Yet, CPLEX using Algorithm 3 is 10% faster than CPLEX with its default crash procedure on instances for which CPLEX needs more than 1000 s to solve.

Finally, the proposed algorithms lend themselves to speculative parallelization of the simplex algorithm. With respect to the dual CPLEX with default initialization, the proposed algorithms lead to speedups of 1.2 and 1.3 on two and four cores, respectively.

References

1. Al-Najjar, C., Malakooti, B.: Hybrid-LP: finding advanced starting points for simplex, and pivoting LP methods. Comput. Oper. Res. **38**, 427–434 (2011)
2. Amestoy, P.R., Davis, T.A., Duff, I.S.: An approximate minimum degree ordering algorithm. SIAM J. Matrix Anal. Appl. **17**, 886–905 (1996)
3. Andersen, E.D., Andersen, K.D.: Presolving in linear programming. Math. Program. **71**, 221–245 (1995)

4. Bertsimas, D., Tsitsiklis, J.: Introduction to Linear Optimization. Athena Scientific, Boston (1997)
5. Bixby, R.E.: Implementing the simplex method: the initial basis. ORSA J. Comput. **4**, 267–284 (1992)
6. Carstens, D.M.: Crashing techniques. In: Orchard-Hays, W. (ed.) Advanced Linear-Programming Computing Techniques, pp. 131–139. McGraw-Hill, New York (1968)
7. Chvátal, V.: Linear Programming. W. H. Freeman, New York (1983)
8. Curtis, A.R., Reid, J.K.: On the automatic scaling of matrices for Gaussian elimination. J. Inst. Math. Appl. **10**, 118–124 (1972)
9. Dantzig, G.B.: Programming in a linear structure. Econometrica **17**, 73–74 (1949)
10. Dantzig, G.B.: Linear Programming and Extensions. Princeton University Press, Princeton (1963)
11. Davis, T.A.: Algorithm 915, SuiteSparseQR: multifrontal multithreaded rank-revealing sparse QR factorization. ACM Trans. Math. Softw. **38**, 8–29 (2011)
12. Davis, T.A., Gilbert, J.R., Larimore, S.I., Ng, E.G.: A column approximate minimum degree ordering algorithm. ACM Trans. Math. Softw. **30**, 353–376 (2004)
13. Davis, T.A., Gilbert, J.R., Larimore, S.I., Ng, E.G.: Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm. ACM Trans. Math. Softw. **30**, 377–380 (2004)
14. Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Math. Program. **91**, 201–213 (2002)
15. Elble, J.M., Sahinidis, N.V.: A review of LU factorization in the simplex algorithm. Int. J. Math. Oper. Res. **4**, 319–365 (2012)
16. Elble, J.M., Sahinidis, N.V.: A review of the LU update in the simplex algorithm. Int. J. Math. Oper. Res. **4**, 366–399 (2012)
17. Elble, J.M., Sahinidis, N.V.: Scaling linear optimization problems prior to application of the simplex method. Comput. Optim. Appl. **52**, 345–371 (2012)
18. Erisman, A.M., Grimes, R.G., Lewis, J.G., Poole Jr., W.G.: A structurally stable modification of Hellerman–Rarick's P^4 algorithm for reordering unsymmetric sparse matrices. SIAM J. Numer. Anal. **22**, 369–385 (1985)
19. Forrest, J.J., Goldfarb, D.: Steepest-edge simplex algorithms for linear programming. Math. Program. **57**, 341–374 (1992)
20. Forrest, J.J.H., Tomlin, J.A.: Updated triangular factors of the basis to maintain sparsity in the product form simplex method. Math. Program. **2**, 263–278 (1972)
21. Gilbert, J.R., Moler, C.B., Schreiber, R.: Sparse matrices in MATLAB: design and implementation. SIAM J. Matrix Anal. Appl. **13**, 333–356 (1992)
22. Goldfarb, D.: On the Bartels–Golub decomposition for linear programming bases. Math. Program. **13**, 272–279 (1977)
23. Gould, N.I.M., Reid, J.K.: New crash procedures for large systems of linear constraints. Math. Program. **45**, 475–501 (1989)
24. Gould, N.I.M., Toint, P.L.: Preprocessing for quadratic programming. Math. Program. **100**, 95–132 (2004)
25. Gülpinar, N., Mitra, G., Maros, I.: Creating advanced bases for large scale linear programs exploiting embedded network structure. Comput. Optim. Appl. **21**, 71–93 (2002)
26. Harris, P.M.J.: Pivot selection methods of the Devex LP code. Math. Program. **5**, 1–28 (1973)
27. Huangfu, Q., Hall, J.: Parallelizing the dual revised simplex method. Math. Program. Comput. **10**, 119–142 (2018)
28. Junior, H.V., Lins, M.P.E.: An improved initial basis for the simplex algorithm. Comput. Oper. Res. **32**, 1983–1993 (2005)
29. Kaczmarz, S.: Angenäherte auflösung von systemen linearer gleichungen. Bull. Int. Acad. Pol. Sci. Lett. **35**, 355–357 (1937)
30. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**, 359–392 (1998)
31. Lenstra, J.K., Rinnoy Kan, A.H.G., Schrijver, A. (eds.): History of Mathematical Programming. CWI North Holland, Amsterdam (1991)
32. Luh, H., Tsaih, R.: An efficient search direction for linear programming problems. Comput. Oper. Res. **29**, 195–203 (2002)
33. Markowitz, H.M.: The elimination form of the inverse and its application to linear programming. Manag. Sci. **3**, 255–269 (1957). (Originally at The RAND Corporation, Research Memorandum RM-1452, 1955)

34. Maros, I.: Computational Techniques of the Simplex Method. Kluwer Academic Publishers, Boston (2003)
35. Maros, I., Mitra, G.: Strategies for creating advanced bases for large-scale linear programming problems. INFORMS J. Comput. **10**, 248–260 (1998)
36. Mészáros, C., Suhl, U.H.: Advanced preprocessing techniques for linear and quadratic programming. OR Spectr. **25**, 575–595 (2003)
37. Murtagh, B.A., Saunders, M.A.: MINOS 5.1 User's Guide. Technical report, Department of Operations Research, Stanford University, Stanford, CA (1987)
38. Nabli, H.: An overview on the simplex algorithm. Appl. Math. Comput. **210**, 479–489 (2009)
39. Nabli, H., Chahdoura, S.: Algebraic simplex initialization combined with the nonfeasible basis methods. Eur. J. Oper. Res. **245**, 384–391 (2015)
40. Pan, P.Q.: Linear Programming Computation. Springer, Berlin (2014)
41. Papadimitriou, C., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Dover Publications, Mineola (1998)
42. Ploskas, N., Samaras, N.: GPU accelerated pivoting rules for the simplex algorithm. J. Syst. Softw. **96**, 1–9 (2014)
43. Ploskas, N., Samaras, N.: A computational comparison of scaling techniques for linear optimization problems on a graphical processing unit. Int. J. Comput. Math. **92**, 319–336 (2015)
44. Terlaky, T., Zhang, S.: Pivot rules for linear programming: a survey on recent theoretical developments. Ann. Oper. Res. **46**, 203–233 (1993)
45. Tomlin, J.A.: An accuracy test for updating triangular factors. Math. Program. Study **4**, 142–145 (1975)
46. Yannakakis, M.: Computing the minimum fill-in is NP-complete. SIAM J. Algebr. Discrete Methods **2**, 77–79 (1981)
47. Ye, Y.: Eliminating columns in the simplex method for linear-programming. J. Optim. Theory Appl. **63**, 69–77 (1989)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Nikolaos Ploskas[1] · Nikolaos V. Sahinidis[2] · Nikolaos Samaras[3]

✉ Nikolaos V. Sahinidis
sahinidis@cmu.edu

Nikolaos Ploskas
nploskas@uowm.gr

Nikolaos Samaras
samaras@uom.gr

1 Department of Electrical and Computer Engineering, University of Western Macedonia, 50100 Kozani, Greece

2 Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA

3 Department of Applied Informatics, University of Macedonia, 54636 Thessaloniki, Greece