INTERNATIONAL TRANSACTIONS IN OPERATIONAL RESEARCH

Intl. Trans. in Op. Res. 10 (2003) 53–64

Performance evaluation of a family of criss-cross algorithms for linear programming

Tibérius Bonates and Nelson Maculan

Universidade Federal do Rio de Janeiro, Programa de Engenharia de Sistemas e Computação – COPPE-UFRJ, Caixa Postal 68511 21145-970 – Rio de Janeiro - RJ - Brasil
E-mail: {tiberius,maculan}@cos.ufrj.br

Received 11 April 2002; received in revised form 30 July 2002; accepted 8 August 2002

Abstract

In this article we evaluate a family of criss-cross algorithms for linear programming problems, comparing the results obtained by these algorithms over a set of test problems with those obtained by the simplex algorithms implemented in the XPRESS[1] commercial package. We describe the known criss-cross variants existing in the literature and introduce new versions obtained with a slight modification based on the original criss-cross algorithm. Moreover, we consider some computational details of our implementation and describe the set of test problems used.

Keywords: Linear programming, simplex method, criss-cross algorithms

1. Introduction

The first criss-cross algorithm was published in 1969 by Stanley Zionts (1969). At that time, there was great attention over the simplex method, that had proved to be successful in solving a number of real-life linear programming problems. However, there was a difficulty in applying the simplex method to some problems: to find a feasible initial basis for starting the algorithm. Although the algorithm itself was not able to find a feasible basis, it was shown to be possible to find such a basis for a specific problem applying the simplex algorithm to a modified version of this problem (Chvátal, 1983). This procedure, known as the *first phase* of the simplex algorithm, returns a feasible basis from which the algorithm can be started. It has been widely used in linear programming solvers, shown to be effective in practice.

We consider the following problem as the standard linear programming problem (LP). The algorithms will be described based on this definition:

[1] XPRESS is a trademark of Dash Optimization, Inc. (http://www.dashoptimization.com/)

(P)

$$min \quad c^\top x \tag{1}$$

$$subject\ to: \quad Ax = b \tag{2}$$

$$x \geqslant 0, \tag{3}$$

where $A \in \mathbb{R}^{m \times n}$, with $rank(A) = m$, $x \in \mathbb{R}^n$ and real vectors c and b of appropriate dimensions.

Unfortunately, to find a feasible initial basis for a linear programming problem is a hard computational task (Bixby, 1992; Enge and Huhn, 1997). For some problems the first phase of the simplex algorithm can take most of the total time spent to solve a problem. In fact, the first phase procedure (*i.e.*, to find a feasible basis) is theoretically equivalent to solving the problem (Enge and Huhn, 1997).

The first criss-cross algorithm was an attempt to avoid the first phase procedure and solve the problem starting from any bases. While the simplex algorithm starts in a feasible vertex and walks along the edges of feasible space, the criss-cross algorithm of Zionts starts at any bases and walks through infeasible vertices, alternating primal and dual iterations, until it arrives at a feasible vertex. The primal iterations are made applying a primal simplex pivot rule to a primal feasible subproblem of the original problem, obtained by removing some infeasible rows of the problem. Similarly, the dual iterations are made applying a dual simplex pivot rule to a dual feasible subproblem, obtained by removing some infeasible columns of the problem. Once a (primal or dual) feasible vertex is acheived, the algorithm of Zionts reduces to the (primal or dual, respectively) simplex algorithm. This may seem to be the first phase for the simplex method, but some relevant differences can be realized:

1. Although we consider only a fraction of the problem in each iteration, there is no modification in the problem's structure;
2. From its start the algorithm is directly searching for the optimal basis and not for a feasible basis. When we use a first phase procedure we change the objective function in some way to drive the algorithm in the search for a feasible basis. In fact, the main hope of Zionts' algorithm is to find the optimal solution during such criss-cross iterations;
3. Reduction to a simplex-type algorithm is not specific as in two-phase simplex algorithms: the two-phase primal (respectively dual) simplex algorithm reduces to the standard primal (resp. dual) simplex algorithm when a feasible basis is found. In the criss-cross context, the choice of which algorithm must be used after feasibility is achieved depends on the nature of the feasible solution encountered (primal or dual).

For a detailed description of the original criss-cross algorithm we refer to Zionts (1969), in which the author introduces his method.

Some years later, another paper was published by Zionts (1972) concerning the empirical behavior of his algorithm. The author made a comparison between an implementation of the criss-cross algorithm and similar implementations of the standard primal and dual simplex algorithms. According to Zionts the criss-cross algorithm was shown to be very efficient in solving the test problems, presenting better results than the standard primal and dual simplex methods. In general, the criss-cross algorithm required a smaller number of iterations and solved the problems faster than the standard simplex algorithms.

Despite the good results related by Zionts, the need for a more rigorous evaluation of the empirical performance of the criss-cross algorithm is evident. The test problems used by Zionts were problems with very small dimension (at most 25 rows or columns). Moreover, a visible weakness of Zionts' algorithm is its convergence proof. The convergence guarantee was based on the insertion of a regularization constraint, followed by a reduction to one of the simplex algorithms, if a large number of criss-cross iterations were made without finding a feasible basis.

At that time, it was not clear if one could propose a finite criss-cross pivot rule, i.e., a criss-cross pivot rule that could solve the problem (or report it as infeasible or unbounded) in a finite number of iterations, without invoking one of the simplex algorithms.

2. The criss-cross variants

Terlaky (1985) proposed a finite version of the original criss-cross algorithm of Zionts to the setting of oriented matroid programming. Fukuda and Matsui (1991) proved that Terlaky's algorithm was also finite in the linear programming case. Since Terlaky's algorithm could be started from an arbitrary bases and could visit infeasible vertices of the problem, it is considered the first finite criss-cross variant, i.e., the first algorithm that could solve a linear program in one single phase based on Zionts' idea of following an external path outside the feasible space during the solution process.

In fact, the same algorithm was developed almost at the same time by three different researchers in different contexts (Bonates, 2001). But, for simplicity, it is commonly cited as Terlaky's algorithm or as the general criss-cross scheme. The algorithm consists of a least index pivot rule applied to the problem. Among the infeasible (primal or dual) variables, the one with least index is chosen to drive the pivoting operation. For this reason it is also called the least index criss-cross algorithm (LICC).

We need to make some definitions before introducing Terlaky's algorithm. These definitions will also hold for the entire text. Let $\mathscr{B}$ be a basis for (P), $\mathscr{N}$ be the non-basis set and $A_{\mathscr{B}}$, $A_{\mathscr{N}}$, $c_{\mathscr{B}}$, $c_{\mathscr{N}}$, $x_{\mathscr{B}}$, $x_{\mathscr{N}}$ the submatrices and subvectors of A, c and x indexed by $\mathscr{B}$ and $\mathscr{N}$, respectively. Let us also define $\tilde{c} = (c_{\mathscr{N}}^{\top} - c_{\mathscr{B}}^{\top} A_{\mathscr{B}}^{-1} A_{\mathscr{N}})$, $\tilde{A} = A_{\mathscr{B}}^{-1} A_{\mathscr{N}}$ and $\tilde{b} = A_{\mathscr{B}}^{-1} b$. Whenever we refer to $\tilde{b}_i$, $\tilde{c}_j$ and $\tilde{a}_{ij}$ as components of $\tilde{b}$, $\tilde{c}$ and $\tilde{A}$, we are in fact, referring to those components of $\tilde{b}$, $\tilde{c}$ and $\tilde{A}$ that are associated with the variables of $\mathscr{B}$ and $\mathscr{N}$ indexed by i, j and (i, j), respectively. According to these conventions, we can describe Terlaky's algorithm as follows in Figure 1. The *pivot* (p, q) statement means a pivoting operation that makes the p-th variable enter the basis and the q-th variable leave.

Unlike the original criss-cross method, Terlaky's algorithm works in a single phase, which means that the algorithm follows the same pivot rule from its start until its end. Therefore, it does not reduce to one of the simplex algorithms when a feasible solution is reached.

Fukuda and Matsui (1991) also showed that the pivot rule used by Terlaky could be relaxed, showing that it was possible to devise new pivot rules for the criss-cross algorithm. In fact, some years later, two new variants of the criss-cross algorithm were proposed by Zhang (1997). Zhang showed that the criss-cross method under a *last in first out/last out first in* (LIFO/LOFI) rule and under a most-often-selected-variable (MOSV) rule was still finite. The variants of Zhang were very similar to Terlaky's algorithm: there was no need for a first phase procedure and they could follow a path through infeasible points before solving the problem.

LICC Algorithm

1. Let $\mathcal{I} = \{i \in \mathcal{B} : \tilde{b}_i < 0\}$.
2. Let $\mathcal{J} = \{j \in \mathcal{N} : \tilde{c}_j < 0\}$.
3. If $\mathcal{I} \cup \mathcal{J} = \emptyset$ then stop: *current basis is optimal.*
4. Let $p = min\{\mathcal{I} \cup J\}$.
5. If $p \in \mathcal{I}$ then go to step **7**.
6. Else go to step **10**.
7. Let $q = min\{j \in \mathcal{N} : \tilde{a}_{pj} < 0\}$.
8. If does not exist such a q then stop: *primal problem is infeasible.*
9. Else `pivot(`q`,`p`)` and go to step **1**.
10. Let $q = min\{i \in \mathcal{B} : \tilde{a}_{ip} > 0\}$.
11. If does not exist such a q then stop: *primal problem is unbounded.*
12. Else `pivot(`p`,`q`)` and go to step **1**.

Fig. 1. Least index criss-cross algorithm.

MOSVCC Algorithm

1. Let $\mathcal{I} = \{i \in \mathcal{B} : \tilde{b}_i < 0\}$.
2. Let $\mathcal{J} = \{j \in \mathcal{N} : \tilde{c}_j < 0\}$.
3. If $\mathcal{I} \cup \mathcal{J} = \emptyset$ then stop: *current basis is optimal.*
4. Let $p =$ `mosv`$\{\mathcal{I} \cup J\}$.
5. If $p \in \mathcal{I}$ then go to step **7**.
6. Else go to step **10**.
7. Let $q =$ `mosv`$\{j \in \mathcal{N} : \tilde{a}_{pj} < 0\}$.
8. If does not exist such a q then stop: *primal problem is infeasible.*
9. Else `pivot(`q`,`p`)` and go to step **1**.
10. Let $q =$ `mosv`$\{i \in \mathcal{B} : \tilde{a}_{ip} > 0\}$.
11. If does not exist such a q then stop: *primal problem is unbounded.*
12. Else `pivot(`p`,`q`)` and go to step **1**.

Fig. 2. Most-often-selected-variable criss-cross algorithm.

We show in Figure 2 the most-often-selected-variable criss-cross algorithm of Zhang. We use a function `mosv` S to indicate the operation of retrieving the index of the most often selected variable from the set S.

The LIFO/LOFI rule is very similar to the MOSV rule. The selection is made choosing the variable that was most recently chosen to be pivoted. Both the algorithms are finite. The finiteness proof of the LIFO/LOFI algorithm is shown in Zhang (1997), while the details of the finiteness proof of the MOSV algorithm are described in Bonates (2001).

The known finite criss-cross variants are not able to recognize a feasible (primal or dual) solution, unless it is optimal (i.e., both primal and dual feasible). The reason for this is that the criss-cross variants do not perform any kind of *ratio test*, as is done in simplex algorithms to preserve feasibility (Chvátal, 1983). These algorithms can enter and leave the primal and dual feasible spaces several times before finding an optimal basis or detecting infeasibility/unboundedness of the problem. As a consequence, the nature of the points generated during the solution process may vary noticeably. If we find a feasible point far from the optimal solution, we may need to use just a few criss-cross iterations to achieve another feasible point next to the optimal solution. The same idea makes us think that starting from an infeasible basis these algorithms would be able to find the optimal solution following an external path, without visiting feasible vertices. This behavior suggests that a shorter path could be

followed in the search for an optimal basis if we use a criss-cross algorithm instead of a simplex-type algorithm.

3. Preliminary computational experiments

To evaluate the relative performance of the criss-cross algorithms, in comparison to the simplex algorithms, we implemented the original algorithm of Zionts and the finite criss-cross variants of Terlaky and Zhang using the XPRESS Optimization Subroutine Library (XOSL) (Dash Optimization, 2001). XOSL is a collection of routines provided by XPRESS to call internal functions of the solvers embedded in the package and to retrieve and set the data related to a linear programming problem, like the matrix and vectors that define the problem as well as the current basis, the inverse of the current basis matrix, the reduced costs, etc. The routines of XOSL provide a reliable environment to handle these data in a relatively simple way without dealing with the details of the underlying operations involved.

The criss-cross algorithms were implemented exactly as described in the papers of Zionts (1969), Fukuda and Matsui (1991) and Zhang (1997). The simplex algorithms used for performance comparison were the revised primal and dual simplex algorithms available in the XPRESS package. The default parameters were kept: the Big-M method is used in the primal algorithm and the pivot rules are the usual pivot rules adopted by the XPRESS solvers. The same solvers and parameters were used to implement Zionts' algorithm, since it needs to invoke one of the simplex algorithms.

The set of test problems we used includes the test problems proposed by Zionts, when he made the first practical evaluation of his algorithm (Zionts, 1972), and some randomly generated problems. In Appendix A, we describe the formulation of these test problems and how to generate them.

When performing the tests with the finite criss-cross variants a common behavior was detected. The algorithms required a very large number of iterations to solve the problems. Even when a feasible solution was achieved during early iterations, the algorithms used to escape from the feasible space as fast as they entered it. These variants tended to enter and leave the (primal or dual) feasible space several times before finding the optimal solution. The number of infeasible columns (infeasible dual variables) and the number of infeasible rows (infeasible primal variables) reflect this behavior. During the run of the algorithms these numbers vary widely. An example of a typical execution of these algorithms is shown in Figure 3. In a given iteration, if the current point is primal feasible, it is usually very far from dual feasibility and *vice-versa*. We can also see in Figure 3 that even if a primal feasible point is found, the following iteration may take the algorithm to a primal infeasible point. In fact, we realized that in general the algorithms leave the (primal or dual) feasible space in no more than three iterations after entering it.

In spite of the finiteness of the criss-cross variants, the tests revealed the need for a great number of iterations and a prohibitive computation time to solve the test problems. The large number of feasible and infeasible bases, even for problems of small dimensions, and the absence of a ration test device to preserve feasiblity are good reasons to explain such great amounts of time and iterations.

The algorithm of Zionts also presented a bad performance. Although it uses the reduction to the simplex algorithms when feasibility is present, the total number of iterations needed to find the optimal solution was usually too large. In all the problems tested, we noticed that no feasible point was found before the maximum number of criss-cross iterations was reached. We defined this number as

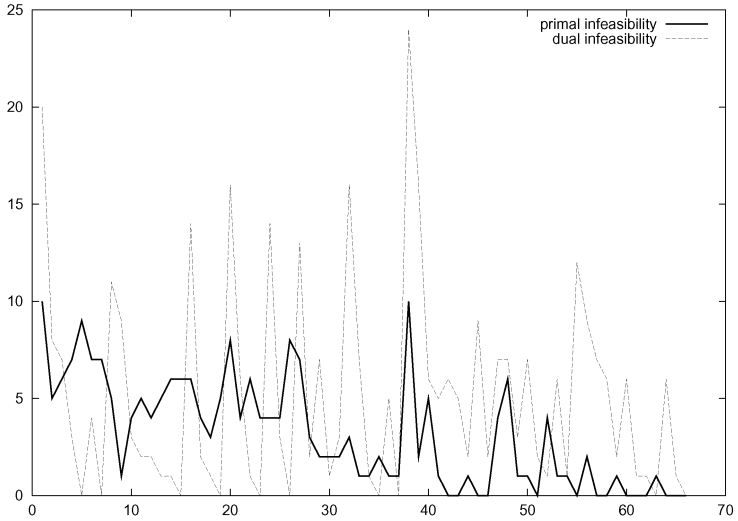

Fig. 3. Typical behavior of finite criss-cross variants.

$2 \times (m + n)$, exactly as was used by Zionts in its experiments. Unfortunately, we did not find the same results as Zionts. We present a more detailed comment on the performance of Zionts' algorithm in subsection 5.4.

4. Modified criss-cross variants

The criss-cross variants we implemented presented a weak performance compared to the performance of the primal and dual simplex algorithms. Analysing the behavior of the criss-cross iterations in terms of primal and dual feasibility levels, we can imagine that the use of a device capable of keeping these criss-cross algorithms near or even inside the feasible space would be very interesting. Such a device could reduce significantly the number of iterations required by these algorithms.

A modification that could ensure the feasibility of the points generated during the iterations that follow the first iteration in which a feasible point is found, would be equivalent in reducing these variants to a simplex-type algorithm under a specific pivot rule. For example, if we find a (primal or dual) feasible solution in a given iteration of a criss-cross variant, it would be enough to add a *ratio test* in the algorithm, keeping the same pivot rule, to force the (primal or dual) feasibility of the solutions generated at subsequent iterations. This is, in fact, the same idea as the one used in Zionts' algorithm, that reduces to an appropriate simplex algorithm in the presence of feasibility. The main difference is that in Zionts' algorithm only a subset of the rows or columns are candidates for being chosen in the pivot selection. In our modified variants, all the rows and columns are candidates.

The finiteness of these modified variants depends on the finiteness of both the criss-cross variant and the simplex algorithm used. Let us suppose that we are using one of the known finite criss-cross variants and that we apply a finite (primal or dual) simplex rule when a feasible basis is found. We know that a finite number of criss-cross iterations is enough to solve a given problem or to prove its

unboundedness/infeasibility. If, at a given iteration, we find a feasible basis, we stop the use of criss-cross iterations and apply a primal or dual simplex rule, according to the type of basis found. The finiteness of the simplex algorithm applied ensures that, in this case, we will also have a finite number of iterations until the problem is solved or reported as unbounded/infeasible.

We implemented these modifications for the LICC, the LIFO/LOFI and MOSV criss-cross algorithms. This was implemented in a way similar to Zionts' algorithm, using the same parameters for the simplex algorithms of XPRESS. In the next section we present a computational evaluation of these algorithms and compare their behavior with the behavior of the simplex algorithms provided by XPRESS.

5. Performance evaluation

We used two sets of problems to evaluate the relative performance of the algorithms implemented. The details of the formulation of these problems are described in Appendix A.

Below, in Figures 4 and 5, we show the names and the numbers of lines and columns of each of the problems tested. In Figure 4, we have problems with inequality (*Zionts*1) and equality (*Zionts*2) constraints. In Figure 5, we show the randomly generated problems, with the additional information of the density of each problem.

The random problems tested are infeasible. We chose also to evaluate the performance of these algorithms in recognizing the infeasibility of a problem, since this is an important question in some methods that make use of linear solvers, such as branch-and-bound algorithms.

There was some difficulty in choosing the test problems. We tried to find problems for which an infeasible starting basis was readily available. A feasible starting basis would not be interesting, since the modified variants would reduce immediately to the simplex algorithms, that would make

Instance	Lines	Columns
pz1_1	250	500
pz1_2	500	750
pz1_3	750	1000
pz1_4	1000	1500
pz1_5	1500	2000

(a) Instances of type *Zionts*1

Instance	Lines	Columns
pz2_1	250	500
pz2_2	500	750
pz2_3	750	1000
pz2_4	1000	1500
pz2_5	1500	2000

(b) Instances of type *Zionts*2

Fig. 4. Instances proposed by Zionts.

Instance	Lines	Columns	Density (%)
pab_1	250	500	2.0
pab_2	500	750	1.0
pac_1	750	1000	1.0
pac_2	1000	1500	1.0

Fig. 5. Randomly generated instances.

any comparison impossible. For the random problems that we used, and also for Zionts' problems, the canonical basis always provided an infeasible basis from which we could start the algorithms without any further calculations.

We applied the algorithms that we implemented, as well as the simplex algorithms of XPRESS, to each of these problems. In Figures 5 and 6 we summarize the results obtained in terms of number of iterations and average execution time (over three executions) with a Pentium II 500MHz, with 256MB RAM. Figure 6 shows the number of iterations required by each algorithm to solve each problem (or to declare it as infeasible). For the criss-cross modified variants we show, in each entry, the number of *pure* criss-cross iterations needed to find a feasible solution, followed by the number of simplex iterations necessary to find the optimal solution. In Figure 7, we have the average execution time, in

Instance	Zionts	LICC	LIFOCC	MOSVCC	XP-PS	XP-DS
pz1_1	—	5/ 465	10/ 1667	10/ 1667	443	500
pz1_2	—	8/1057	7/ 3315	7/ 3315	1271	1065
pz1_3	—	5/1862	8/ 6831	18/ 6831	2294	1726
pz1_4	—	7/2894	—	12/10300	3671	2834
pz1_5	—	7/5169	—	6/13257	5976	4892
pz2_1	—	7/ 486	—	—	883	434
pz2_2	—	8/1176	—	—	2052	1108
pz2_3	—	9/2124	—	—	3684	2134
pz2_4	—	10/3624	—	—	6010	3336
pz2_5	—	—	—	—	—	—
pab_1	—	134/0	15/0	514/0	379	267
pab_2	—	7/0	39/0	453/0	801	1121
pac_1	—	54/0	—	395/0	2546	4822
pac_2	—	—	—	121/0	2735	12260

Fig. 6. Number of iterations of each algorithm.

Instance	Zionts	LICC	LIFOCC	MOSVCC	XP-PS	XP-DS
pz1_1	—	3,30	13,33	13,93	3,06	3,15
pz1_2	—	25,89	74,89	75,24	30,46	22,60
pz1_3	—	95,07	611,05	610,34	113,28	83,17
pz1_4	—	276,24	—	2161,68	379,01	270,05
pz1_5	—	1147,70	—	—	1403,19	1053,05
pz2_1	—	4,18	—	—	10,94	3,47
pz2_2	—	44,93	—	—	99,83	38,33
pz2_3	—	178,40	—	—	452,09	180,75
pz2_4	—	655,48	—	—	1547,15	603,69
pz2_5	—	—	—	—	—	—
pab_1	—	5,87	0,34	25,35	0,15	0,10
pab_2	—	0,44	3,03	47,61	0,59	1,09
pac_1	—	9,19	—	88,49	5,21	11,67
pac_2	—	—	—	44,67	16,90	172,30

Fig. 7. Execution time of each algorithm.

seconds, of each algorithm when applied to each problem. In both tables we use the symbol '—' to denote that a given algorithm was not able to solve that problem, or to find a feasible basis for it after 30 minutes, which means that the algorithm worked poorly for that specific problem.

The results of the original variants are not shown in the tables because these algorithms were outperformed by the simplex algorithms by very large differences of execution time and number of iterations. As one can see, the algorithm of Zionts also presented low efficiency in solving these problems.

We can see that the modified variants behaved differently for each type of problem. For most of the problems, however, the LICC algorithm presented good results. Its overall performance was better than the performance of the other variants and in some cases it was even better than the simplex algorithms.

5.1. Zionts*1 instances*

All the *Zionts*1 instances are problems with finite optimal solution. The LICC algorithm performed better than all the other variants, achieving results as good as the results obtained by the primal and dual simplex algorithms. In fact, the LICC algorithm solved almost all the problems (with one exception) in less time and requiring a smaller number of iterations than the dual simplex method. Its performance was similar to the primal simplex algorithm. In some cases, the number of iterations required by LICC was lower than the number required by the primal simplex.

The MOSV algorithm solved all the problems, but showed very poor efficiency, spending more time and more iterations than the simplex algorithms. The LIFO criss-cross algorithm was not able to solve all the problems within the time limit. The two large problems required more than 30 minutes to be solved. The other problems were solved with results that were similar to the results of the MOSV algorithm. The Zionts' algorithm could not finish its execution within the time limit for any of the instances.

5.2. Zionts*2 instances*

When dealing with the *Zionts*2 instances, the algorithm of Zionts, the MOSV and LIFO/LOFI criss-cross variants were unable to solve the problems in a reasonable time. On the other hand, the LICC algorithm outperformed the primal simplex algorithm for all the problems, showing very good results, with less time and iterations than the dual simplex for some problems.

Despite the existence of equality constraints in these problems, they are all feasible, with finite optimal solution. These constraints make more difficult the task of finding a feasible basis. This characteristic seemed not to affect the performance of the LICC algorithm. Analyzing the results, we believe that the LICC algorithm could be a good alternative when dealing with problems of this nature.

5.3. Random instances

We realized different results when treating random problems. The LICC algorithm, that performed much better than the other variants for the Zionts' problems, showed a small number of iterations and required good execution times, but could not finish its execution in a reasonable time for one of the random problems. The MOSV algorithm declared the infeasibility of all the problems in a very small number of iterations. Unfortunately, the time spent was much longer than the time required by the

simplex algorithms of XPRESS. The LIFO algorithm required a small number of iterations and a reasonable time, but was unable to deal with problems of larger dimensions.

It is interesting to remember that only criss-cross iterations were used for these problems, since they do not admit any feasible basis. In general, a small number of criss-cross iterations were needed to prove the infeasibility of these problems.

5.4. The performance of Zionts' algorithm

It is clear that the original algorithm of Zionts performed badly, without finishing its execution within the time limit for any of the problems tested. The primal and dual infeasibility levels (i.e. the number of primal and dual infeasible variables) vary widely during its execution, explaining the poor efficiency of the algorithm. In Figure 8 we can see a typical behavior of the primal and dual infeasibility levels. The x-axis refers to the iterations, while the y-axis reflects the number of infeasible variables.

The algorithm does not improve the infeasibility levels from iteration to iteration, neither does it present a tendency of doing this along the execution, which means that it finds a feasible basis almost by 'accident'. As a consequence, the feasible basis obtained by the algorithm is usually very far from the optimal solution. The number of simplex iterations needed to solve the problem starting from the basis obtained was usually larger than the number of iterations required by the simplex algorithms to solve the problem from scratch.

This difference in performance between our tests and the Zionts' tests may be explained in terms of the different implementation of the simplex algorithms used in each case. The simplex algorithms used by Zionts were the standard two-phase simplex algorithms, while the algorithms available within the XPRESS package are the revised simplex versions (Chvátal, 1983). The primal simplex of XPRESS uses a Big-*M* strategy to find a feasible basis, thus, the simplex algorithms used by Zionts were much less sophisticated than the current implementations available in the XPRESS package.

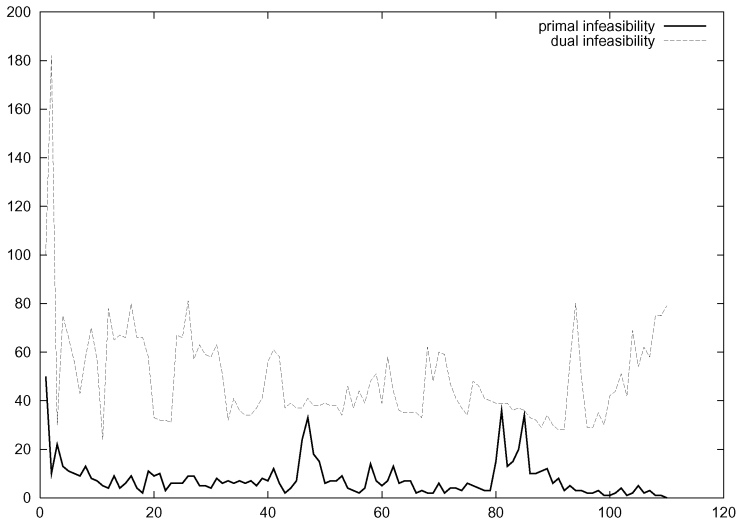

Fig. 8. A typical behavior of Zionts' algorithm.

6. Conclusions

As one can see, the finite criss-cross variants – as well as the modified variants proposed here – do not use the information corresponding to the magnitude of the components $\tilde{c}_j$, $\tilde{b}_i$ and $\tilde{a}_{ij}$. In fact, these algorithms are usually called combinatorial pivoting algorithms because they use only the signs of those components to decide which variables should be pivoted. The main reason for this is that the existing criss-cross pivot rules arose primarily in the oriented matroid programming context, being extended to the linear programming case later.

Pivot rules for the criss-cross algorithms that use not only the signs, but also the absolute values of $\tilde{c}_j$, $\tilde{b}_i$ and $\tilde{a}_{ij}$, should be called non-combinatorial rules. One example of such rules for the simplex method is the known steepest-descent rule. Unfortunately, non-combinatorial criss-cross pivot rules are not proposed yet and it is still not clear if one can propose finite non-combinatorial rules for the general criss-cross scheme.

The use of combinatorial pivot rules frequently leads us to simple finiteness proofs. This is the case for Terlaky's least index criss-cross algorithm and also for the variants that arose later. It seems that the same cannot be said about non-combinatorial rules. To prove the finiteness of non-combinatorial criss-cross rules seems to be not so easy, since we do not have the strong argument of the ordering of the variables. Also, there is no evident way to guarantee the maintenance or improvement for the feasibility levels or for the value of the objective function from iteration to iteration.

Such non-combinatorial criss-cross variants would probably show better performance than the ingenuous combinatorial versions. We could also devise even more efficient modified variants, incorporating some geometrical insight into these rules, what could drive the algorithms to the optimal solution faster or improve the quality of the feasible bases provided by the criss-cross phase, that are usually far from an optimal basis.

The results shown in this paper are preliminary, but can be interpreted as good evidence that the criss-cross algorithms can be used in practice and that there is still an open field for investigating non-combinatorial pivot rules for these algorithms.

7. Acknowledgements

We would like to thank the Brazilian National Council for Scientific and Technological Development (CNPQ) for the financial support provided during the development of this research. We also thank the support of PRONEX, CTPETRO and FUJB.

References

Bixby, R., 1992. Implementing the Simplex Method: The Initial Basis, *ORSA Journal on Computing* 4(3): 267–284.

Bonates, T., 2001. *Implementação do Algoritmo Criss-Cross para Problemas de Programação Linear de Grande Porte (in Portuguese)*, Master's thesis, COPPE, Federal University of Rio de Janeiro, Rio de Janeiro, Brazil.

Chvátal, V., 1983. *Linear Programming*, W.H. Freeman and Company, New York.

Dash Optimization, 2001. *XPRESS-MP: User Manuals*.

Enge, A. and Huhn, P., 1997. A Counterexample to H. Arsham's 'Intialization of the Simplex Algorithm: an Artificial-free Approach', *SIAM Review* 39: 736–744.

Fukuda, K. and Matsui, T., 1991. On the Finiteness of the Criss-Cross Method, *European Journal of Operations Research* 52: 119–124.

Terlaky, T., 1985. A Convergent Criss-Cross Method, *Math. Oper. und Stat. ser. Optimization* 16(5): 683–690.

Zhang, S., 1997. New Variants of Finite Criss-Cross Pivot Algorithms for linear programming, *Technical Report 9707/A*, Econometric Institute, Erasmus University Rotterdam.

Zionts, S., 1969. The Criss-Cross Method for Solving Linear Programming Problems, *Management Science* 15(7): 426–445.

Zionts, S., 1972. Some Empirical Tests of the Criss-Cross Method, *Management Science* 50(19): 406–410.

Appendix: Test problems

In this appendix we describe the formulation of the set of test problems used to evaluate the algorithms. The problems proposed by Zionts (1972) follow the form below:

$$\begin{aligned} \textit{max} \quad & z = \sum_{j=1}^{n} c_j x_j \\ \textit{subject to}: \quad & \sum_{j=1}^{n} a_{ij} x_j \leqslant b_i,\ i = 1, \ldots, m \\ & x_j \geqslant 0,\ j = 1, \ldots, n. \end{aligned}$$

For the *Zionts*1 problems, the components c_j and b_i are generated according to the following convention:

$$c_j = \begin{cases} -1, \text{ when } j \text{ is odd;} \\ +1, \text{ when } j \text{ is even,} \end{cases} \qquad b_i = \begin{cases} +10\,000, \text{ when } i \text{ is odd;} \\ -100, \text{ when } i \text{ is even.} \end{cases}$$

The a_{ij} components are uniformily distributed within the interval [1, 1000] whenever i is odd, and within the interval $[-1, -1000]$ otherwise.

The *Zionts*2 problems are similarly generated. The only differences are the constraints with $i = 1, 4, 7, 10$ and so on, always skipping 2 constraints. These are equality constraints with the components a_{ij} uniformily distributed within the interval [1, 1000] and $b_i = 100$:

$$\sum_{j=1}^{n} a_{ij} x_j = 100,\ i = 1, 4, 7, 10, \ldots$$

Although there is no guarantee regarding the feasibility of the *Zionts*2 problems, all the test problems we generated were feasible problems with finite optimal solution.

As one can see, in the problems proposed by Zionts all the entries of A are non-zero, *i.e.* the matrix has density equal to 100%. Problems of such type are unlikely to appear in practical applications. So, we decided to generate some low density problems, in order to evaluate the algorithms when applied to problems of this type too. For this purpose we implemented a random generator of problems with specific density.

The random problems have all the components c_j and b_i uniformily generated within the interval $[-1, +1]$. The non-zero components a_{ij} (randomly selected) are also generated within the same interval. Clearly, the feasibility of these random problems cannot also be ensured *a priori*. In fact, as we said, all the random problems we generated were infeasible.