Reaching Feasibility Quickly for Sets of Linear Constraints

by

Vladislav Brion

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Carleton University

Ottawa, Ontario

Abstract

Finding a feasible solution for a set of linear constraints and bounds is an essential step in many problems, including linear programming. Many linear systems have large numbers of variables and constraints, and the computation time for finding a feasible point can be very large. This thesis proposes an improved projection method for finding a feasible point in linear systems. The method increases the acceleration and improves the direction of movement towards the feasible region. Multiple algorithms developed in this research apply various approaches for the movement acceleration. The thesis also develops an optimization model for finding an optimal set of algorithms having the highest performance in a concurrent implementation. A new presolving technique is introduced which simplifies a linear system before starting the main algorithms. Concurrent computing of the algorithms is proposed for finding a feasible point in large linear systems.

Acknowledgements

I would like to express my hearty gratitude to Professor John Chinneck for his assistance in all stages of this research. I want to thank the professors and other support staff from the Department of Systems and Computer Engineering at Carleton University for their help. I want to thank my wife and my children for the inspiration they gave me.

Table of Contents

List of Figures

List of Tables

List of Acronyms

BA	Basic Acceleration
CC	Constraint Consensus
DBavg	Average-Direction Based
DBmax	Maximum Direction Based
FDfar	Feasibility Distance Far
GD	Generalized Direction
GDM	Matrix of generalized directions
GDA	Generalized Directions Algorithm
GDRI	Generalized Direction Row Index
LP	Linear Programming
MAUD	Maximum Acceleration of Unique Direction
MPS	Mathematical Programming System
TANM	Tangent Matrix
TCI	Tangent Column Index
TUDA	Tangent Unique Direction Acceleration

List of symbols

α feasibility distance tolerance

β angle between feasibility vectors

γ angle between a movement vector and constraint

δ angle between constraints

μ maximum number of iterations

λ relaxation parameter

A matrix of constraint coefficients

a_{ij} element of matrix A

$\boldsymbol{b}$ vector of constants

$\boldsymbol{c}$ vector of coefficients of objective function

D combination of problems

dir indicator of direction

$\boldsymbol{e}$ vector of elastic variables

fd feasibility distance

$\boldsymbol{fv}$ feasibility vector

g multiplier

$iter$ number of iteration

I nimber of algorithms

J number of problems

K number of initial points

$\boldsymbol{l}$ vector of lower bounds

M number of cases

m number of constraints

$maxtan$ maximum value of values in a row of tangent matrix

$meantan$ average of values exceeding 1 in tangent matrix

n number of variables

n_{viol}, $nviol$ number of violated constraints

n^s	size of the minimum set of algorithms
p	number of processors (cores)
pf	power of feasibility vector
pow	power of multiplier
P_Q	projection onto set Q
q	coefficient of proportion
qp	number of initial points for acceptable outputs
r	ration of feasibility distances at previous and current iteration
s	size of set of algorithms
$s(p)$	speedup
$sumg$	sum of multipliers g
T_p	parallel computing time
T_s	sequential computing time
$\boldsymbol{t}$	consensus (movement) vector
tol	tolerance
$\boldsymbol{u}$	vector of upper bounds
V	matrix of scores
$\boldsymbol{v}$	normalized perpendiculars
vc	vector of counts of consecutively violated constraints
$\boldsymbol{X}$	binary vector for optimization
$\boldsymbol{x}$	variable vector (point) to be determined
z	index of column type

1. Introduction

Solving a system of linear constraints is a well-known problem. Its standard form is:

$$Ax \leq b \tag{1.1}$$

where:

- $x \in R^n$ is a variable vector to be determined;
- A is an $(m \times n)$ matrix of constraint coefficients;
- b is an $(m \times 1)$ vector of constants;

In general, all linear systems can be converted to this standard form. For example, constraints with a "≥" sign are converted to the standard form by multiplying through by -1. An equality constraint is transformed to a pair of constraints with the same coefficients and different signs (≥ and ≤).

The variables may have bounds as follows:

$$l_j \leq x_j \leq u_j \tag{1.2}$$

where l_j and u_j are a lower and an upper bound for the variable x_j. In such cases the bounds can be considered as constraints and converted to the standard form.

A multidimensional region defined by (1.1) and (1.2) is called a *feasible region*. Under linear constraints, a feasible region consists of the intersection of half planes, and is therefore always convex. Any point $\boldsymbol{x}$ that satisfies all of the constraints (1.1) and (1.2) is called a *feasible point*, and it is located in a feasible region. If a feasible region is empty, and the corresponding point $\boldsymbol{x}$ does not exist, a system defined by (1.1) and (1.2) is called *infeasible*.

Finding a feasible point is an essential initial step in many problems, including solving linear programs. Linear programming is used in many areas, such as operational research, economics, industry and engineering. Below are examples of such problems [24]:

- Allocation models where resources should be optimally allocated under restricted cost.
- Blending models where ingredients should be optimally mixed under various requirements, such as restricted supply.
- Operation planning models where amounts of various types of production should be optimized in order to increase profit which is the difference between income from sales and costs of materials and labor.
- Finding maximum flow in networks.

Projection methods for finding feasibility which will be introduced in Section 3, also have various applications [4], [6], [7], [8] as follows:

- Radiation therapy treatment planning.
- Image reconstruction.

In general, the number of constraints and variables in (1.1) can be very large, therefore quickly finding a feasible point can be challenging. As a result, a common drawback of existing algorithms for finding a feasible point is large computing time. Moreover, some methods produce a point which is too far from the feasible region.

There are two main types of algorithms for finding a feasible point:

- Exact algorithms which have a mathematically proven solution. The drawback is possibly extremely large computing time for systems with very high numbers of constraints and

variables. For example, the simplex method [26] and the interior point method [19] are algorithms whose exact solutions were proved.

- Approximate (heuristic algorithms) which produce an approximate solution (a point close to a feasible region). It is difficult to evaluate their theoretical performance and complexity. In many cases the execution time of heuristic algorithms is estimated via simulation. However, they can provide an approximate solution for large-size problems in a shorter time than exact algorithms.

A large class of heuristic methods are projection algorithms. These are iterative algorithms using projections of various types onto sets [5] for iterating the point location. Many algorithms for finding a feasible point for linear and non-linear systems are projection methods.

A subclass of projection methods are those for finding a feasible point in convex regions [8], such as successive orthogonal projections, block-iterative projections, etc. Some of these methods use a projection onto a single set at a time, whereas others employ simultaneous projections on multiple sets, and apply their combinations to update the point for the next iteration.

A significant advantage of projection methods is that they can be implemented using various architectures of parallel computing. Modern computers contain multiple cores and can take advantage of this characteristic. A simple example is simultaneous projection methods where projections onto various groups of sets are computed at the same time on different cores.

The objective of this thesis is to develop fast algorithms for finding feasible solutions for large sets of linear constraints. Projection methods are ideal for this task because of their suitability for concurrent implementation.

2. Background

2.1 Linear Programming

A Linear Programming (LP) model is an optimization problem where a linear objective function $c^T x$ must be optimized subject to a set of linear constraints (1.1) and (1.2). If a feasible point is unknown, it should be determined before starting the optimization step in some of the main LP solution algorithms. Some LP problems may not have an objective function and need only a feasible solution.

The most-used LP algorithm is the Simplex method [26]. From its invention 70 years ago, it has been improved multiple times to increase its performance. It uses a property based on the convexity of the feasible region and the linearity of the objective function: if the value of an objective function at a vertex of the feasible region is bigger than the values at all of its neighbouring vertices then this vertex maximizes the objective function. At each iteration, vertices connected to the current one are checked, and a "better" vertex (having a better value of an objective function compared with one at the previous iteration) is selected. The algorithm stops if there are no better vertices.

The algorithm starts at an arbitrary point. If it is not feasible, a first stage using the ideas described above is applied for finding a feasible vertex of a feasible region. The simplex method has a non-polynomial worst-case complexity which can cause very large computing times for high-dimensional problems.

There are multiple other methods for LP. The Khachian method [24] uses multidimensional ellipsoids which move towards an optimal point. The prune-and search algorithm [3] is based on evaluating a fraction of the constraints (hyperplanes) which are parallel to the vector of the

objective function. Such constraints are redundant and are pruned away. Both of the algorithms have very high complexity for problems with large numbers of variables.

Karmarkar's interior point method [19] uses movements across the feasible region. This method usually needs few iterations, but the complexity of each iteration is very high. The interior point method belongs to the class of projection methods mentioned below.

Most of the aforementioned algorithms can start only from a feasible point, which demonstrates the importance of quickly finding a feasible point.

2.2 Accessory Algorithms

2.2.1 Introduction

Most projection algorithms are iterative methods that start from an initial point which can be selected in various ways. To prevent infinite computation, all iterative algorithms should have at least one stopping condition, such as a maximum number of iterations and/or maximum computing time.

The following tasks can accelerate finding a feasible point:

- Presolving.
- Intelligent selection of the initial point.
- Concurrent implementation.

2.2.2 Presolving

Many linear systems can be simplified before applying a main algorithm for finding a feasible point. Some simple examples of the possible problem simplifications follow:

- Infeasible problem. A pair of constraints such as $x_1 + 2x_2 \leq 5$ and $2x_1 + 4x_2 \geq 15$ will never be satisfied. Any optimization algorithm should not be started because it will not produce a feasible result.
- Redundant constraint. If the first constraint from a pair $x_1 + 2x_2 \leq 5$ and $x_1 + 2x_2 \leq 10$ is satisfied, the second one will always be satisfied, and can be removed.
- Tightening of constraints and bounds. For example, a constraint $x_1 + x_2 \leq 8$ with the bounds $x_1 \geq 3$ and $x_2 \geq 5$ can be satisfied only if $x_1 = 3$ and $x_2 = 5$. In this case the variables can be fixed and the constraint can be removed. There may be a cascade of subsequent simplifications.

Presolving is a set of methods which analyses interactions between constraints and can simplify the set of constraints before applying a further optimization algorithm. The main presolving cases and techniques, such as singleton rows/columns, fixed variables, forcing and dominated constraints/columns, were described by Andersen and Andersen [2]. The possible simplifications performed by the presolving tasks are:

- Reducing the number of constraints or tightening the bounds. This makes the main algorithm run faster due to a smaller number of constraints and bounds to analyze.
- Fixing variables has multiple effects. It is not only reduces the number of variables to be analyze, it can also simplify or even eliminate other constraints containing the fixed variables. Moreover, if the set of constraints has a very narrow feasible region, finding a feasible point can be difficult; fixing the variables may eliminate this problem.
- Finding an infeasible set of constraints during presolving means that the entire time needed for the main algorithm is not needed.

The set of presolving tasks can be applied iteratively until no further simplifications are possible. However, the presolving tasks can be time-consuming, especially when a single interaction is produced by large number of constraints. Therefore, the presolving time or the number of presolving iterations.

2.2.3 Selecting an Initial Point

Intelligent selection of the initial point can significantly decrease the computing time needed for finding a feasible point. In the best case an initial point is feasible, and no further computation is needed, but this is rare, especially for high-dimensional problems, because the probability of simultaneously satisfying all constraints is very low. However, the variable space can be reduced by selecting initial values for all dimensions within their bounds (if they exist), thus preventing bound violations.

There are three main methods to select an initial point within the variable bounds for linear and non-linear systems [20]. The simplest method is random sampling within the variable bounds. The other two methods, stratified sampling and Latin hypercube sampling, are based on selecting variables from subintervals of the variable space [20]. For each dimension these subintervals can have various sizes that allow selecting desired areas for sampling. As an example, if subintervals are based on a logarithmic scale, most of the generated values will be closer to the lower bounds.

The variable space can be extremely large because many problems do not have finite upper bounds, or values of existing bounds are very large. Some solvers [21] set an initial value closest to zero if the corresponding variable has at least one infinite bound.

2.2.4 Parallel and Concurrent Computing

Computing time can be significantly reduced when parallel computing is used instead of sequential computation, if the algorithm is suitable. For example, matrix multiplication can be simultaneously performed on multiple nodes when all of them perform computations on different blocks of matrices. Platforms for parallel programming can have different architectures for executing instructions/data streams, memory, and interconnection [16]. Evaluating performance of parallel systems is based on comparison of the sequential runtime T_s and the parallel runtime T_p which strictly depends on the number of processors p. This performance is described by a value of speedup which can be calculated as follows:

$$s(p) = T_s / T_p \tag{2.1}$$

Assume that the time needed for interconnection between processors is negligible. Then:

$$1 \leq s(p) \leq p$$

The maximum value of speedup, called linear speedup, occurs when the entire code of an algorithm is executed in parallel. For most algorithms, a part of their code must be executed sequentially, and the value of the speedup will not reach its maximum.

Concurrent computing is similar to parallel computing, except that the computations do not proceed in lockstep: algorithms can start and terminate at different times. Various cores can apply different algorithms on different data, such as different initial points.

2.3 Measurement of infeasibility

If a point violates a single constraint, the level of violation can be expressed by the shortest Euclidian distance between a point and a single violated constraint. Chinneck [10] defined this

distance as the *feasibility distance*. For a violated linear constraint it is calculated by the following expression:

$$fd_i = \frac{(\boldsymbol{a_i x} - b_i)}{\sqrt{\sum_{j=1}^{m} a_{ij}^2}} \tag{2.2}$$

If $fd_i > \alpha$, where α is a feasibility distance tolerance, the constraint *i* is violated.

The shortest vector to move a point to satisfy violated constraint *i* is defined as the *feasibility vector* $\boldsymbol{fv_i}$ which can be found as follows:

$$\boldsymbol{fv_i} = \frac{-fd_i \boldsymbol{a_i^T}}{\sqrt{\sum_{j=1}^{m} a_{ij}^2}} \tag{2.3}$$

Both of these metrics are normalized, and, therefore, are not sensitive to a row scaling.

When a point is infeasible, at least one constraint is violated. The distance between a point and a feasible region can be evaluated by following metrics:

- Maximum feasibility distance among all violated constraints;
- Sum of the feasibility distances over all violated constraints;
- Number of violated constraints.

The Euclidean distance between an infeasible point and a feasible region is the L^2- norm, whereas the maximum feasibility distance is the L^∞- norm, and the sum of the feasibility distances is the L^1-norm. Therefore, each of these three values is an approximate evaluation of the distance from a point to a feasible region.

3. Finding Approximate Feasible Solution for Linear Systems: State of the Art

3.1 Projection Methods

Projection methods are heuristic algorithms. They are iterative methods using projections onto convex sets in different ways [5]. Projection methods were originated by von Neumann in 1933 [23], Kaczmarz in 1937 [18], and later have been developed by Censor and others [5, 6, 9]. The common steps of these methods are selecting an arbitrary initial point and iterating its next position using relaxations of the projections [8]. A large group of projection methods is the row-action methods [4] which treat violated constraints sequentially. A well-known method from this group is the method of successive orthogonal projections which can be applied to linear and non-linear systems. It is based on the following iteration step:

$$x^{k+1} = x^k + \lambda_k (P_{Q_{i(k)}}(x^k) - x^k),$$

where $P_{Q_{i(k)}}(x^k)$ is the orthogonal projection of the current point x^k onto the set $Q_{i(k)}$ representing a single constraint, and λ_k is a relaxation parameter used for increasing or decreasing the vector which moves the point to its next position.

The orthogonal projection is equivalent to the feasibility vector from the point to the constraint. Projection of an infeasible point onto a single violated constraint determines the movement direction to the closest feasible point [27] which is a feasibility vector. For linear inequalities with $\lambda_k \geq 1$, the constraint will be satisfied at iteration $k + 1$.

Cimmino [13] introduced a simultaneous projection method where a point is projected to all hyperplanes at the same time. A linear combination of the individual orthogonal projections onto

single constraints is used to update the point. Various algorithms based on his method can be implemented for parallel computing [7].

Various algorithms based on the method of orthogonal projections can be applied for problems with linear constraints. Gould [15] applied this method with different relaxation coefficients λ_k for finding feasible points for multiple Netlib LP problems, the standard set of linear programming test models. It is most commonly used for testing software for solving sets of linear constraints. Some of the problems were solved relatively quickly, whereas others resulted in large errors after 1 million iterations.

Chubanov [11] proposed an extended projection method using projections not only onto original constraints, but also onto additional artificial constraints. His method is applicable for finding both feasible and optimal solutions. The polynomial complexity of the algorithm was proved. Basu et al [1] evaluated the exact polynomial complexity of the Chubanov's method for finding a feasible point. A main disadvantage of Chubanov's method is that it can be applied only for positive values of **x**. In addition, it has a relatively high degree of polynomial complexity, which further increases for larger ranges of **x**.

Censor developed multiple optimization methods based on projection methods. His ART3+ method [6] is not a fully simultaneous projection method. Only one pair of constraints is used at a single iteration. This method shows better results than those obtained by Gould [15], but it is applicable only for problems with inequality constraints. Another method is linear superiorization [9] which simultaneously treats both the optimality and feasibility problems. The resulting feasible point has a relatively better value of the objective function. The disadvantage of this algorithm is that it deals only with positive values of **x**.

Censor [7] further developed component averaging methods based on simultaneous projection methods. Similar to all such methods, each component of the movement vector is calculated as a weighted average of the corresponding variables in the violated constraints. However, only the violated constraints containing these variables are considered. For example, for a pair of violated constraints $x_1 \leq 5$ and $x_1 + 2x_2 \leq 10$, the second component is based only on the second constraint.

3.2 Constraint Consensus methods

The basic Constraint Consensus (CC) method was developed by Chinneck [10] for nonlinear systems, but it also can be applied for linear systems. It is a component averaging method

Inputs:

- a set of *m* constraints in *n* variables x_j
- an initial point $\boldsymbol{x}^0$
- a feasibility distance tolerance α
- a maximum number of iterations μ

1. Initialize: $\boldsymbol{x} \leftarrow \boldsymbol{x}^0$
2. Repeat μ times:
   2.1. $n_{viol} \leftarrow 0$; for all *j*: $n_j \leftarrow 0$; $s_j \leftarrow 0.0$
   2.2 For each constraint *i*:
      2.2.1. Calculate feasibility distance fd_i from $\boldsymbol{x}$ to constraint *i*.
      2.2.2. If $fd_i > \alpha$:
         2.2.2.1. $n_{viol} \leftarrow n_{viol} + 1$;
         2.2.2.2. Calculate feasibility vector $\boldsymbol{fv}_i$ from $\boldsymbol{x}$ to constraint *i*:
         2.2.2.3. For each variable *j* in constraint *i*: $n_j \leftarrow n_j + 1$; $s_j \leftarrow s_j + fv_{ij}$;
   2.3. If $n_{viol} = 0$: exit successfully;
   2.4. For each variable *j*:
      2.4.1. If $n_j > 0$: $t_j \leftarrow {}^{s_j}/_{n_j}$; else: $t_j \leftarrow 0$.
   2.5. $\boldsymbol{x} \leftarrow \boldsymbol{x} + \boldsymbol{t}$;
   2.6. If any x_j exceeds its bound: reset onto its violated bound.
3. Exit unsuccessfully.

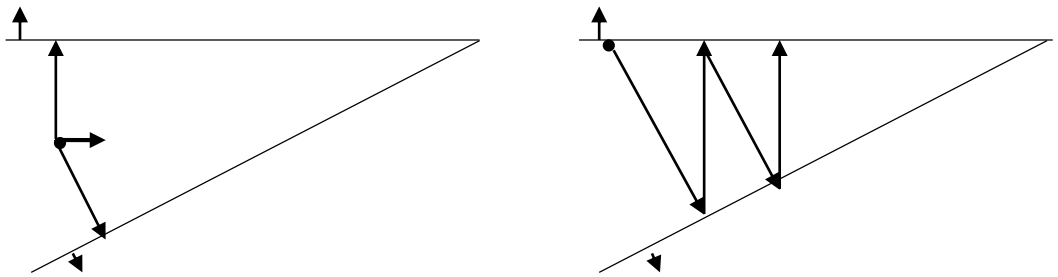
Algorithm 3.1 Basic CC algorithm

producing a resulting movement called the *consensus vector* $\boldsymbol{t}$. This vector is a component-wise

average of the feasibility vectors for the violated constraints. Algorithm 3.1 shows the basic CC algorithm.

The algorithm has a flaw, inherent in many simultaneous projection methods: an interaction of single projections (feasibility vectors) may produce a very short resulting movement vector. This problem may repeat for many iterations, causing very poor performance.

Figure 3.1 shows two cases of inefficient movements:



a) Ill-conditioned case b) Zigzag

Figure 3.1 Two cases of inefficient movements

Chinneck [10] called the case shown in Figure 3.1a the ill-conditioned case when the length of the movement vector is very short because the angle between a pair of constraints is small. Figure 3.1b shows another example of an inefficient movement: poor direction of the movement vectors at multiple iterations creates a zigzag pattern. This happens because only one constraint from a pair is violated at each iteration, and an angle between them is small.

Ibrahim and Chinneck [17] developed multiple improved CC methods by setting different weights for violated constraints. The weights can depend on the movement directions (*DBavg*, *DBmax*) or on the distance to a violated constraint (*FDfar*). The improved methods are less sensitive to ill-conditioning, but also can produce zigzag movements. The improved methods are faster, but still slow, and can be improved.

4. Faster Solution of Large Systems of Linear Constraints

4.1. Motivation

Many modern heuristic algorithms for finding a feasible point for systems of linear constraints are based on simultaneous projection methods. Most of the methods have performance limitations and/or have limited applications. For example, methods such as Chinneck [10] and Gould [15] show the following drawbacks when they are applied to large linear systems:

- Slow convergence after a large number of iterations and long running time.
- When convergence is not reached after a large maximum number of iterations, the obtained point is still far from a feasible region.

An algorithm of Censor [6] has higher performance than that of Chinneck [10] and Gould [15]. However, it is applicable only to problems with inequality constraints.

The polynomial algorithm of Chubanov [11] has high polynomial complexity and increases with larger ranges of $\boldsymbol{x}$.

Therefore, finding a feasible point for large linear systems of constraints can be improved by finding a quicker method which will be applicable to all forms of linear systems. This will allow an earlier start of the second stage of the LP problem, resulting in a faster overall solution. In addition, simply finding a feasible solution is often the goal.

4.2. Problem statement

The goal of this thesis is to develop a new method for quickly finding an approximately feasible point for large systems of linear constraints. The method should be applicable to

problems with both inequality and equality constraints, and should be implementable for concurrent computing.

4.3 Overview of the Thesis

This thesis proposes the following main improvements of the existing CC methods:

1. *Presolving*. Andersen and Andersen [2] described multiple presolving techniques for use when finding an optimal point. When finding a feasible point only, the objective function can be disregarded, and further reductions, aimed only at achieving feasibility, are possible.
2. *Modified movement acceleration*. As stated in Section 3, Chinneck [10] reported very short lengths of the movement vector for ill-conditioned pairs of constraints. A new approach uses a multiplier of various components of feasibility vectors depending on the angle between violated constraints. As a result, the lengths of shorter movement vectors are increased more than the lengths of longer movement vectors.
3. *Step back with further expansion*. This method prevents zigzag movements. A partial step back can improve the accuracy of the direction of the next movement vector, and its length is increased using analysis of the improvement of the violated constraints.

To evaluate the last two improvements, the performance is compared to the existing CC algorithms developed by Ibrahim and Chinneck [10]. The presolving innovation can be used at the start of any method, and is tested separately.

The new CC method has a large number of possible settings based on different approaches to the movement acceleration, resulting in multiple algorithms. The performance of each of the algorithms is tested using sequential computing. An optimization model is developed to find the

best minimum set of the algorithms for concurrent processing. Simulation of concurrent processing of this set of algorithm variants suggests that it provides the best performance compared to other sets of algorithms.

In addition, this research analyses the efficiency of selecting a special prespecified initial point.

5. Improved Algorithms for Fast Approximate Solution of Linear Systems

5.1 Introduction

The new CC algorithms improve the movement vectors for the ill-conditioned cases and zigzags shown in figure 3.1. In general, the stages of finding a near-feasible point are as follows:

1. Transforming the system to the standard form (1.1). All constraints are transformed to "≤" type.
2. Executing the presolving tasks.
3. Transforming finite lower and upper variable bounds to constraints and adding them to the existing constraints. After this step the original variable bounds may be used only for selecting an initial point. As a result, the same method is used to find feasibility vectors to both violated constraints and violated bounds.
4. Selecting an initial point and executing the main algorithm.

The improvements developed in this research do not increase the complexity of a single CC iteration which is $O(m \times n)$. The expected computing time of a single iteration in the new algorithms will be larger than in the existing ones, but the overall time will be reduced by a significant reduction in the number of iterations.

Implementation of the algorithms is based on sparse matrix structures. However, for simplicity of explanation, all algorithms are described here using dense rectangular matrix structures.

5.2 Presolving

The main presolving techniques for LP, described by Andersen and Andersen [2] are applicable for finding a feasible point. However if the objective function is disregarded, as we can when seeking only feasibility, there is scope for additional model reductions. This section describes a new technique related to uniform-sign columns that can be used instead of the column singleton technique. Given that there is no objective function, and all constraints have $\leq$ type, columns having coefficients with a uniform sign can be forced to their lowest possible value, thereby giving the greatest help in satisfying the constraints. This effect is obtained by setting the variable to its bound (lower or upper, depending on the sign of the coefficients). In addition, if the corresponding bound is large, the constraints having non-zero coefficients in this column will always be satisfied, and can be removed from the model. Otherwise, the variable is removed and the corresponding values of the vector *b* are updated. After removing column *j* from the matrix *A*, some rows can become empty, having only zero coefficients, and can be removed. The definition of a uniform-sign column, assuming that at least one of the coefficients is not zero, is the following:

$$\exists(j)\text{: } \exists(i)\ a_{ij} \neq 0 \text{ AND } (\forall i \in 1 \dots m, \leq 0 \text{ OR } \forall i \in 1 \dots m, a_{ij} \geq 0) \qquad (5.1)$$

For presolving, all infinite variable lower and upper bounds are replaced by values -1e10 and 1e10 correspondingly, and called *large values*. Presolving a uniform-sign column is shown in Algorithm 5.1.

1. For each column $j \in 1 \ldots n$ in the matrix A:
    1.1. If all coefficients in column j are non-positive:
        1.1.1 Set the variable j to its upper bound u_j.
        1.1.2 For all rows i that have a nonzero coefficient in column j:
            1.1.2.1 If the upper bound is large: remove the row.
            1.1.2.2 Else: adjust RHS as follows: $b_i \leftarrow b_i - a_{ij} \times u_j$.
        1.1.3 Remove column j.
    1.2. If all coefficients in column j are non-negative, apply steps 1.1.1 - 1.1.3 using lower bound l_j instead of upper bound u_j.
    1.3. Remove empty rows from A, if any exist.

Algorithm 5.1. Presolving task for a uniform-sign column

The possible benefits of the presolving procedure are the following:

1. Tightening bounds of constraints and variables. The movement vector will be more precise.
2. Removing redundant constraints. This action can correct the relative weights of constraints that should slightly improve the movement vector. Reducing the number of constraints will decrease the running time, while leaving the number of iterations unchanged.
3. Removing variables will increase the algorithm performance in two ways:
    - As the number of variables decreases, each iteration of the algorithm will be faster. The number of iterations remains the same, and the improvement of performance depends on the proportion of removed variables to existing variables.
    - When a feasible region is very narrow in various dimensions, long movement vectors can pass over it during many iterations. Removing variables can cancel the narrowness in the corresponding dimensions. This prevents multiple overpasses, which can significantly decrease the number of iterations. This effect can be more significant than faster execution of a single iteration.

An example where the new algorithms may not be efficient without this presolving task are shown in the next section.

After removing the columns having coefficients with uniform sign, and removing constraints thereafter, other columns can achieve uniform-sign coefficients. Therefore, testing columns for uniform-sign coefficients and their possible removal can be done multiple times.

As was mentioned above, a set of presolving tasks should be applied repetitively until they show no improvement. In this work, some of the presolving techniques, such as singleton row, fixed variable, unique-sign column and forcing constraint are applied repetitively to the original Netlib problems. For some problems, continuous improvements occurred during more than 60 iterations. Despite the potential benefits obtained from presolving, the computing time required should be taken into account.

5.3 Tangent Acceleration

The length of the movement vector in an ill-conditioned case can be increased. In other words, an acceleration is applied to this vector. A new acceleration method is proposed in this research that increases the length of various feasibility vectors in different ways. Let δ be the angle between a pair of violated constraints, and $\beta = 180^{\circ} - \delta$ be the angle between their feasibility vectors, as shown in Figure 5.1. The value of both angles cannot exceed 180°.

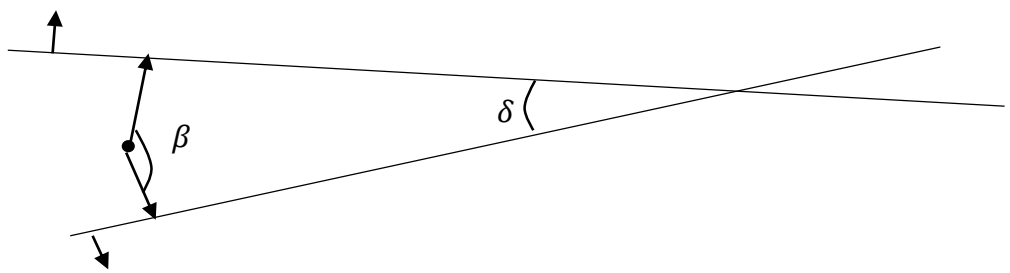


Figure 5.1 Angles between constraints and between feasibility vectors

While δ is decreasing, the movement vector becomes shorter, whereas if $\delta > 90^{\circ}$ the length of the movement vector is sufficient, i.e. the length increase is not needed. Hence, a proposed multiplier for the length of the movement vector is a function of δ or β. The following simple function can be used for the multiplier g:

$$g = \begin{cases} tan\left(\frac{180^{\circ}-\delta}{2}\right) = tan\left(\beta/2\right), \beta > 90^{\circ} \\ 1 \qquad \text{otherwise} \end{cases} \tag{5.2}$$

The value of g at $\delta \geq 90^{\circ}$ is 1. When δ decreases, the pair of constraints becomes more ill-conditioned, and the shorter length of the movement vector is multiplied by a larger value of g. When δ is near 0°, the value of g is very large. A degenerate case of $\delta = 0^{\circ}$ occurs for a range of constraints when both of them cannot be simultaneously violated.

An example of acceleration is shown in Figure 5.2. All new CC algorithms have the acceleration step. Multiple acceleration methods described below have common inputs and symbols which are equivalent to the existing Basic CC algorithm.

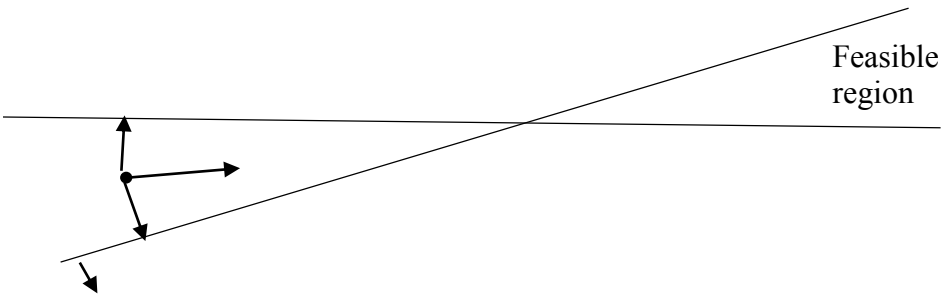


Figure 5.2 Acceleration of movement vector

A high-dimensional problem with a large number of constraints is much more complicated. There are many pairs of constraints, and the following cases should be considered:

- At least one constraint in a pair is satisfied. This pair is not ill-conditioned, and an acceleration for this pair is not needed.
- Both of the constraints in a pair are violated, but this pair is "overridden" by other dominating pairs of constraints, as shown in Figure 5.3, where the pair 1-2 is overridden by the pairs 1-3 and 2-3. Finding an overridden pair of constraints would significantly increase the complexity of an iteration, and it is not considered in the new CC algorithms.
- A dominating pair of violated constraints may need acceleration of the movement vector.

The proposed acceleration part of the new CC algorithms considers all pairs of violated constraints which can be overridden or dominating. For each violated constraint, a set of acceleration coefficients between it and all other violated constraints is calculated. The maximum value in this set is used for acceleration of the feasibility vector for this constraint.

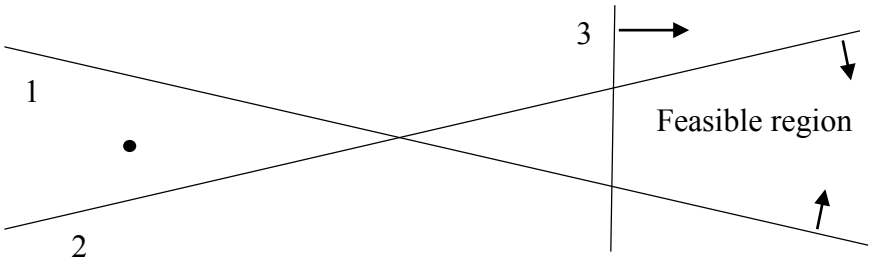


Figure 5.3 Only pairs 1-3 and 2-3 are dominating

For linear systems, a tangent coefficient for each pair of constraints is a constant. As a result, all pairwise tangent coefficients can be calculated at an initialization step. Their values are held in the *tangent matrix TANM*. It is a symmetric matrix with dimensions $m \times m$. Calculation of the tangent matrix is time-consuming because of its complexity $O(m^2 n)$, therefore removing this task from the iterations significantly decreases running time of a single iteration. *The tangent column indexes* (*TCI*) matrix has dimensions $m \times m$ and contains indices of constraints

corresponding to values in the tangent matrix. During initialization the values of each row in the tangent matrix are sorted in decreasing order, and the same row in the *TCI* matrix is permuted correspondingly. In order to find the maximum tangent coefficient for a selected violated constraint *i* at a single iteration, a sequence of the permuted *TCI* indexes in the corresponding row is searched for. When the first index *j* belonging to a violated constraint is found, the corresponding value in the tangent matrix is used for the acceleration coefficient, and the search is stopped, because it is the highest tangent value between a constraint *i* and all other violated constraints with which it intersects. The algorithm this tangent acceleration is called the *basic acceleration* (*BA*) algorithm. It is shown in Algorithm 5.2.

This BA algorithm applies acceleration to the entire feasibility vector. The next section shows how the BA can be modified by applying acceleration to selected components of feasibility vectors.

Inputs:

- $m \times m$ tangent matrix (*TM*) with sorted rows.
- $m \times m$ *TCI* matrix with permuted indices.
- a set *v* with *nviol* indices of violated constraints.

1. For each violated constraint v_i, *i* = 1… *nviol*:
    1.1. Set default acceleration $g \leftarrow 1$.
    1.2. For each permuted index *j* = 1…*n* in the row *i* of *TCI*:
        1.2.1 If constraint *j* is violated:
            1.2.1.1 $g \leftarrow tm_{ij}$ is the acceleration coefficient for the constraint *i*.
            1.2.1.2 Break the loop *j*.
    1.3. Update value of $fd_i \leftarrow fd_i \times g$.

Algorithm 5.2 Basic tangent acceleration

5.4 Improved Tangent Acceleration

Figure 5.2 shows that the components of the movement vector have different importance. The length of the horizontal component should be increased in order to move closer to the feasible region, whereas any lengthening of its vertical component does not significantly improve the movement vector, so vertical acceleration is not needed. If all feasibility vector coefficients in a column *j* have the same sign, then the desired direction of component *j* in the movement vector is straightforward regardless of the other components. If the signs are not uniform, there is no preferred direction for this component. In other words, only selected components of feasibility vectors should be accelerated. As a result, after acceleration the movement vector can change its direction. The acceleration procedure in the following algorithms is based on acceleration of selected components of the movement vector. Its main idea is shown in Algorithm 5.3.

Inputs:

- $nviol \times n$ violation matrix with set of all feasibility row-vectors *fv* for violated constraints.
- All inputs from algorithm 5.2.

1. For each violated constraint *i* = 1… *nviol*:
   1.1. Set default acceleration $g_i = 1$.
   1.2. Select corresponding row in the tangent matrix and find g_i as in algorithm 5.2
   1.3. For every column *j* = 1…*n* in the violation matrix:
      1.3.1. If all coefficients have uniform sign:
         - Update value of $fv_{ij} \leftarrow fv_{ij} \times g_i$.

Algorithm 5.3. Tangent uniform direction acceleration (common description)

This algorithm cannot be efficiently implemented because each *i* and *j* in Step 1.3 is processed in a different way depending on the acceleration coefficient g_i and uniformity of signs in column *j*. However, this column property can be evaluated only at the end of the search over

all rows. As a result, the loop over all rows should be processed twice: to find columns with unique signs and to possibly update fv_{ij} thereafter. As a result, the running time of each iteration can increase significantly. The following two algorithms represent an approximation of Algorithm 5.3. Both use only one search over the rows for finding possible uniformity of column signs in the violation matrix, as well as for calculating the movement vector using the basic CC algorithm. A decision about possible acceleration of its selected components is made at the end of the row search, and the acceleration coefficients are not attributes of a single row.

The first approximation algorithm does not use tangent coefficients for acceleration. It is called the *maximum acceleration of uniform direction* (*MAUD*) algorithm and is shown in Algorithm 5.4. It is equivalent to the basic CC algorithm with added evaluation of the uniform sign columns. At the end of the algorithm, the movement components belonging to the uniform sign columns are replaced by the highest maximum or minimum values of the corresponding components of the feasibility vectors. This algorithm is a modification of the *FDfar* CC algorithm.

Another approximation algorithm is the *tangent uniform direction acceleration* algorithm (*TUDA*), which is similar to Algorithm 5.4, but instead of finding the maximum value of fv_j, it finds an average value of tangents for all violated constraints. Before step 2.1 of Algorithm 5.4,

Variables:

z_j: attribute of column *j*; can have the following values:

$$z_j = \begin{cases} 1, & \text{all currently searched values in column } j \text{ are positive} \\ 2, & \text{all currently searched values in column } j \text{ are negative} \\ -1, & \text{currently searched values in column } j \text{ have different signs} \\ 0, & \text{unknown (after initialization)} \end{cases}$$

1. For each column *j* = 1…*n*:
   1.1. Initialize: $z_j \leftarrow 0$; $s_j \leftarrow 0$; $n_j \leftarrow 0$; $maxfv_j \leftarrow 0$; $minfv_j \leftarrow 0$.
2. For each violated constraint *i* = 1… *nviol*:
   2.1. For every column *j* = 1… *n* in the violation matrix:
      2.1.1. $maxfv_j \leftarrow max(maxfv_j, fv_{ij})$; $minfv_j \leftarrow min(minfv_j, fv_{ij})$
      2.1.2. If $fv_{ij} > 0$:
         2.1.2.1. If $z_j = 1$ *OR* $z_j = 0$: $z_j \leftarrow 1$
         2.1.2.2. Else $z_j \leftarrow -1$
      2.1.3. If $fv_{ij} < 0$:
         2.1.3.1. If $z_j = 2$ *OR* $z_j = 0$: $z_j \leftarrow 2$
         2.1.3.2. Else $z_j \leftarrow -1$
      2.1.4. If $fv_{ij} \neq 0$: $s_j \leftarrow s_j + fv_{ij}$; $n_j \leftarrow n_j + 1$
3. For every variable *j* = 1…*n*:
   3.1. If $z_j = 1$: $t_j \leftarrow maxfv_j$ and break the loop *j*.
   3.2. If $z_j = 2$: $t_j \leftarrow minfv_j$ and break the loop *j*.
   3.3. If $n_j > 0$: $t_j \leftarrow s_j / n_j$; else: $t_j \leftarrow 0$.

Algorithm 5.4. Maximum acceleration of uniform direction

the value of g_i is evaluated as described in Step 1.2 of Algorithm 5.3. Also, the new variable *sumg* is introduced. It is initialized to 0, and before step 2.1 is updated to *sumg* ← *sumg* + g_i. Step 3 is then modified as shown in Algorithm 5.5.

The algorithms of this class can be as slow as the existing basic *CC* algorithm, if there are no columns with unique signs. The possible improvement is a low acceleration of all components of fv_i and an additional acceleration of the selected components.

3. For every variable $j = 1 \ldots n$:
    3.1. If $n_j > 0$: $t_j \leftarrow s_j / n_j$; else: $t_j \leftarrow 0$.
    3.2. If $z_j > 0$: $t_j \leftarrow t_j$ * *sumg / nviol*

Algorithm 5.5 Last part of improved tangent uniform direction acceleration

Another algorithm is an improvement of the *TUDA* and *MAUD* algorithms when the signs of all columns in the violation matrix are nonuniform. This *generalized directions acceleration algorithm* (*GDA*) is based on a criterion called generalized directions. The idea of this method is shown in Figure 5.4.

Let $\overline{v_1}$ and $\overline{v_2}$ be normalized (length 1) perpendiculars to two violated constraints. When $|v_{x1}| > |v_{x2}|$, then moving right is preferable, because the point will be moved towards a feasible region. Similarly, when $|v_{y2}| > |v_{y1}|$, then moving up is preferable. The moves right and up are called generalized movement directions (*GD*) for these constraints, and can be applied to any point violating both constraints. For any feasibility vector, all its components towards the *GD*s are accelerated, whereas the components opposite to the *GD*s are used without acceleration. If, for example, $|v_{x2}| = |v_{x1}|$, this component does not have a *GD*, and the horizontal components of each feasibility vector are not accelerated.

For high-dimensional problems, some *GD*s may not point towards the feasible region. The complexity of their evaluation is high, and it is not considered in this research. All *GD* components are accelerated, and at least one of them will point towards a feasible region. Possible movements in incorrectly accelerated directions will be corrected in later iterations.

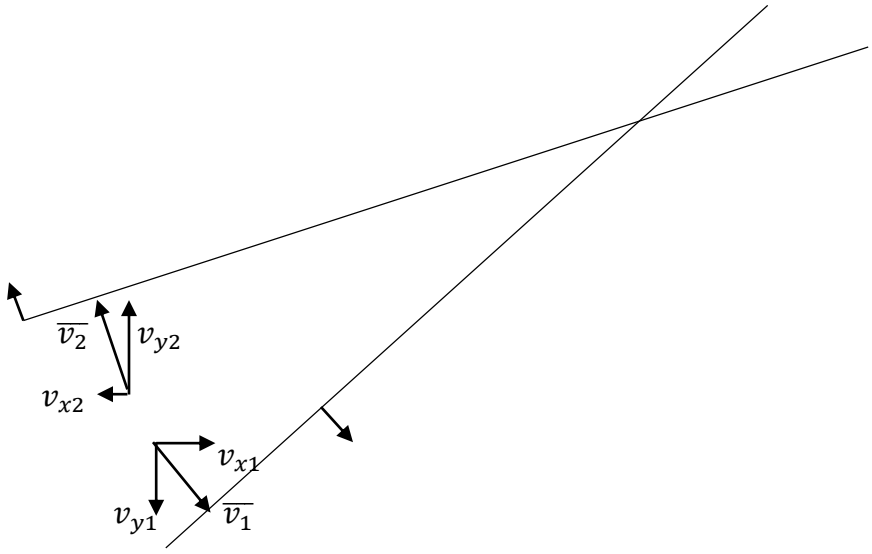


Figure 5.4 Generalized directions

Due to the normalization, the *GD*s do not depend on the point location, and for linear systems they can be calculated in an initialization step and grouped into a matrix. This *GD* matrix has *m* rows and *n* columns. Each row corresponds to a constraint and contains components of the normalized perpendicular to this constraint, whose values can be positive, negative or zero. The following transformations are similar to those for the tangent matrix, but they are applied to columns instead of rows. Another matrix called *generalized direction row indexes* (*GDRI*) is introduced. It also has dimensions $m \times n$ and contains row indexes of each column. At initialization each column of the *GD* matrix is sorted by decreasing order of its absolute values, and the same column in the *GDRI* matrix is permuted correspondingly. The *GD* sorted column values can contain ties, and the sequence of their indexes in permuted *GDRI* columns is arbitrary in this case.

In order to find the indicator of direction *dir* for component *j* in a single iteration, a sequence of the permuted *GDRI* indexes in the corresponding column is searched. When the first index *i* belonging to a violated constraint is found, the corresponding value in the *GD* matrix is selected

because its absolute value is larger than or equal to the unsearched coefficients in this column. If the value is equal to zero, the *GD* for this column does not exist, and $dir_j = 0$. Otherwise, the search is continued only for the same absolute values in this column in order to find possible ties with opposite signs. If the next element in $gd_{.j}$ has lower absolute value, the search is stopped and dir_j is equal to 1 or -1, depending on the sign of the previously selected value. If among the tied values at least one value with opposite sign is found, and its index corresponds to a currently violated constraint, the *GD* for this column does not exist, and $dir_j = 0$.

The *GDA* algorithm is shown in Algorithm 5.6. It needs only one row search, because the condition of accelerating fv_{ij} can be found immediately by comparing its sign with the generalized direction of column *j* which is already known.

Inputs:

- $m \times n$ *GD* matrix with columns sorted by absolute values.
- $m \times n$ *GDRI* matrix with permuted row indices.
- a set *v* with *nviol* indices of violated constraints.
- A tolerance *tol* = 1e-6

1. For each column *j* = 1… *n*:
    1.1. Set default $dir_j \leftarrow 0$; *maxgd* ← 0.
    1.2. For each permuted index *i* = 1…*m* in column *j* of *GDRI*:
        1.2.1 If constraint *i* is violated:
            1.2.1.1 *maxgd* ← GD_{ij} is the value whose absolute value is greater or equal to all other absolute values of component *j*.
            1.2.1.2. $i_1 \leftarrow i$.
            1.2.1.3 Break out of the loop on *i*.
    1.3. If |*maxgd*| > *tol*:
        1.3.1. For each permuted index *i* = i_1+1…*m* in column *j* of *GDRI*:
            1.3.1.1 If |*maxgd*| - |GD_{ij}| > *tol*:
                - $dir_j \leftarrow$ sign(*maxgd*).
                - Break out of the loop on *i*.
            1.3.1.2 If |*maxgd* + GD_{ij}| < *tol* AND constraint *i* is violated:
                - $dir_j \leftarrow 0$
                - Break out of the loop on *i*.

2. Continue Algorithm 2 with Step 1.3.1 modified as follows:
    If ($fv_{ij} > 0$ AND $dir_j = 1$) OR ($fv_{ij} < 0$ AND $dir_j = -1$): $fv_{ij} \leftarrow fv_{ij} \times g_i$.

Algorithm 5.6 Acceleration by generalized directions (GDA)

5.5 Examples

The following simple artificial examples show the improved performance of the new *CC* methods.

Example 1. $x_2 \geq -0.2x_1$; $x_2 \leq 0.2x_1$; $x_1 \leq 5$; $\boldsymbol{x^0}$ = (-4, -0.5)

Methods	Existing *CC*				New *CC*			
	Basic	*FDfar*	*DBavg*	*DBmax*	*BA*	*MAUD*	*TUDA*	*GDA*
Number of iterations	348	238	248	347	166	347	65	65

The three new algorithms converge in fewer iterations, and the two algorithms with acceleration of selected components (TUDA, GDA) are the fastest in terms of a number of iterations.

Example 2. $x_2 \geq 0.5x_1$; $x_2 \leq x_1$; $x_1 \leq 5$; $\boldsymbol{x^0}$ = (-4, -3)

Methods	Existing *CC*				New *CC*			
	Basic	*FDfar*	*DBavg*	*DBmax*	*BA*	*MAUD*	*TUDA*	*GDA*
Number of iterations	524	273	524	524	247	524	524	191

The *GDA* algorithm requires the fewest iterations, because this problem is similar to the one shown in Figure 5.4. If a presolving step updates the lower bound on x_2 to $x_2 \geq 0$, the existing *CC* algorithms will produce better results.

Example 3. $16x_1 - 8x_2 - x_3 \leq 8$; $-16x_1 - 8x_2 - x_3 \leq 8$; $8x_2 - x_3 \leq 0$; $x_3 \leq 5$; $\boldsymbol{x}^0$ = (0, 0.5, 2)

Methods	Existing *CC*				New *CC*			
	Basic	*FDfar*	*DBavg*	*DBmax*	*BA*	*MAUD*	*TUDA*	*GDA*
Number of iterations	2038	1096	1242	626	1106	1381	1150	1287

In this example all of the new *CC* methods perform poorly, because this set is not very ill-conditioned. The maximum coefficients in the tangent matrix are 1.98, and they belong to x_1-x_2 directions, whereas the feasible region is located towards increasing x_3 from an initial point. In other words, the best movement vector contains only a component towards positive x_3-direction. As a result, acceleration is not efficient. However, the variable x_1 is unaffected; it will never change its value because at x_1 = 0 the feasibility distances for the first two constraints will always be the same, and *x* components of their feasibility vectors will cancel each other. The projection of the set onto axis x_1 obtained by setting x_1 to 0 is very ill-conditioned. This example shows that ill-conditioned high-dimensional sets may not be recognized by a tangent matrix. On the other hand, applying presolving tasks for Algorithm 5.1 will remove the variable x_3 having coefficients with uniform sign, and the reduced problem will be solved very quickly. This is a good example of the importance of presolving.

5.6 Tangent Algorithm Variants

5.6.1. Flexible Feasibility Distance Tolerance

The original CC algorithms use a fixed value of the feasibility distance tolerance α for evaluating constraint violation. This research introduces a flexible feasibility distance tolerance described in Algorithm 5.7. During the first *iter* < $\sqrt{n}$ iterations, $\alpha = \alpha_{min}$. After that, the level of α is set to the closest power of 10 which is lower than the current best maximum feasibility distance. When at later iterations the feasibility distances of all (or almost all) constraints are within this tolerance (which can be large), its level is divided by 10, and further points should be satisfy this tighter tolerance. The value of α cannot be decreased below its minimum value (such as 10^{-6}), and when all feasibility distances are within this value, a feasible point is found. Flexible values of α increase the priority of highly-violated constraints, because the less-violated constraints are temporarily considered as satisfied, and will not decrease the length of an average movement vector.

In order to accelerate computation, if the number of violations is not zero, but low enough (2 at Step 2.4), the value of α is updated.

When the value of α is reduced, the iteration is not continued, and a new iteration is started. For a lower value of α, some constraints with unchanged feasibility distances can become violated. At the next iteration all constraints are checked for violation, and the movements are calculated thereafter. Also, before starting a new iteration, the counters activating the relaxation task (Section 5.7) are reset.

If a value of the minimum feasibility distance tolerance α_{min} is a power of 10, Step 2.4 does not permit reducing value of α lower than α_{min}.

Inputs:

- a minimum feasibility distance tolerance α_{min}
- a maximum number of iterations μ

1. Initialize: $\alpha \leftarrow \alpha_{min}$
2. Repeat μ times:
   2.1. Find the current maximum feasibility distance.
   2.2. If the number of iteration is $\lfloor\sqrt{n}\rfloor$:
      2.2.1. Set α to maximum value among α_{min} and the closest power of 10 below the current maximum feasibility distance.
   2.3. If $n_{viol} = 0$ AND $\alpha < 1.001\alpha_{min}$: exit with success.
   2.4. If $n_{viol} \leq 2$ AND $\alpha > 1.001\alpha_{min}$:
      2.4.1. If $n_{viol} = 0$: set maximum feasibility distance to α.
      2.4.2. $\alpha \leftarrow 0.1\alpha$.
      2.4.3. Reset counters for step back task.
      2.4.4. Start the next iteration (step 2).
   2.5. Continue (find feasibility and movement vectors).

Algorithm 5.7 Flexible maximum distance tolerance (part of CC algorithm)

5.6.2 Increasing the Length of the Movement Vector

The value of the multiplier can be increased when g^{pow} ($pow > 1$) is used instead of g. With greater length of the movement vector, the feasible region can be reached faster. On the other hand, if the feasibility vector is not directed towards a feasible region, its larger acceleration can significantly increase the feasibility distance at the next step.

If only selected dimensions, such as uniform sign dimensions and *GD* dimensions are accelerated, they also can be accelerated using by the factor g^{pow}, while the other dimensions are accelerated using a lower factor. In this research g is based on the value of *meantan* which is an average of all factors exceeding 1 in the tangent matrix for all violated constraints.

5.6.3. Additional Improvement of the Movement Direction

It was mentioned in the previous sections that movement vectors may not point at the feasible region when the distances to the violated constraints differ significantly. This problem

mostly occurs with ill-conditioned constraints. The movement direction can be improved by adjusting the feasibility distances. The extended equation for finding a feasibility vector for a violated constraint *i* is following:

$$fv_i = -A_{i.}\, fd_i^{pf} \left(\frac{\sum_{i=1}^{n_{viol}} fd_i}{n_{viol}} \right)^{1-pf} \tag{5.3}$$

The case with $pf = 1$ corresponds to the existing expression of feasibility vectors and total movements. $pf = 0$ corresponds to the extreme case where the length of each feasibility vector does not depend on the corresponding feasibility distance. It is only adjusted by the average of the feasibility distances over all of the violated constraints. A partial adjustment by values of *pf* between 0 and 1 can be applied. An additional option is adjusting only the constraints which are parts of ill-conditioned pairs.

5.6.4. Improving Selected Violated Constraints

The feasibility distance to some constraints may not be improved during a large number of iterations. The simplest way to quickly improve them is by adding priority to a violated constraint in the acceleration step.

1. Initialize vector of counts *vc* of consecutively violated constraints to **0**.
2. At each iteration:
   2.1. Increase counts of all constraints by 1.
   2.2. Reset counts of all satisfied constraints to 0.
   2.3 Find average of all non-zero counts $\overline{vc}$.
   2.4 Multiply the feasibility vector for a violated constraint by $vc_i / \overline{vc}$

Algorithm 5.8 Weights of constraint violations

The priority depends on number of consecutive iterations for which it remains violated. Such adjustment is shown in Algorithm 5.8. Step 2.4 also can be modified by replacing $\overline{vc}$ by 1 in the multiplication.

In this research, a large number of algorithms is obtained by combining different types of tangent methods, various powers for increasing movement length, various powers for changing the movement directions, and using different weights for constraint violations.

5.7. Relaxation

The acceleration of the movement vector described in the previous section improves performance for many ill-conditioned cases. However, in the following cases it is not applied because an ill-conditioned case is not detected:

- A zigzag as shown in Figure 3.2. At each iteration there is only one violated constraint, and the movement vector satisfies it, but causes violation of another constraint. These constraints will not appear as a pair of violated constraints, and the tangent acceleration will not be applied.
- An equality constraint which is converted into a pair of constraints. At most one constraint from the pair can be violated at any particular moment. A movement vector which is accelerated by one of new CC algorithm, can oversatisfy a violated constraint, causing violation of the other constraint in the pair.

If a movement vector does not point towards the feasible region, applying an acceleration may not decrease feasibility distances. Moreover, they can increase. The main problem of poor direction of the movement vector is shown in Figure 5.5.

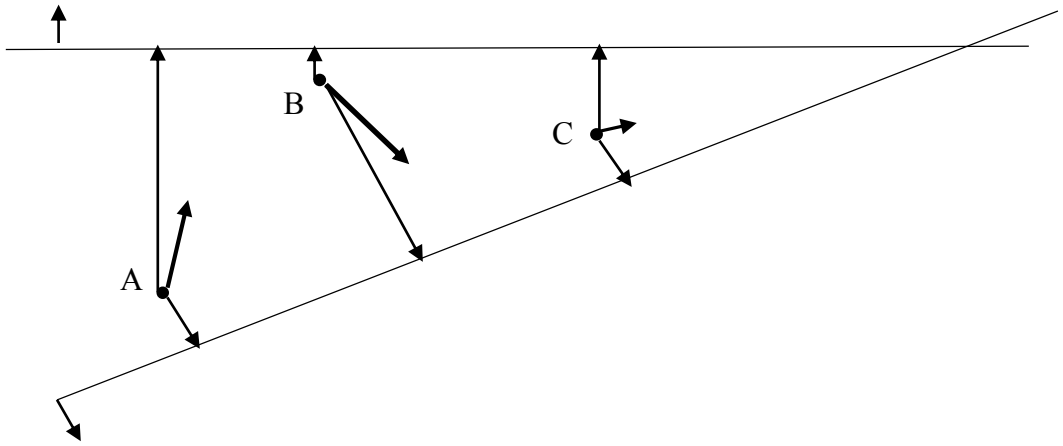


Figure 5.5 Analysis of movement vector directions

All three points A, B and C violate both of the ill-conditioned constraints. The direction of the movement vector obtained from a basic CC algorithm depends on the difference between distances to both violated constraints. The points A and B are near one of the constraints, and the movement vector obtained from the basic CC methods reduces the violation of the more distant constraint. However, in this case the movement vector does not point towards the feasible region. On the other hand, the point C is located at the same distance from both constraints, and its movement vector, if accelerated, will reach the feasible region. Therefore, the best expected direction of a movement vector is obtained under following conditions:

- More constraints should be violated in order to simultaneously impact the movement vector.
- The distances to these constraints should not be significantly different.

The adjustment shown in Section 5.6.3 can partially improve the direction of the movement vector. Another method, which allows finding the point C having the best direction for the movement vector, is shown in Figure 5.6. It is implemented in a separate step called *relaxation*.

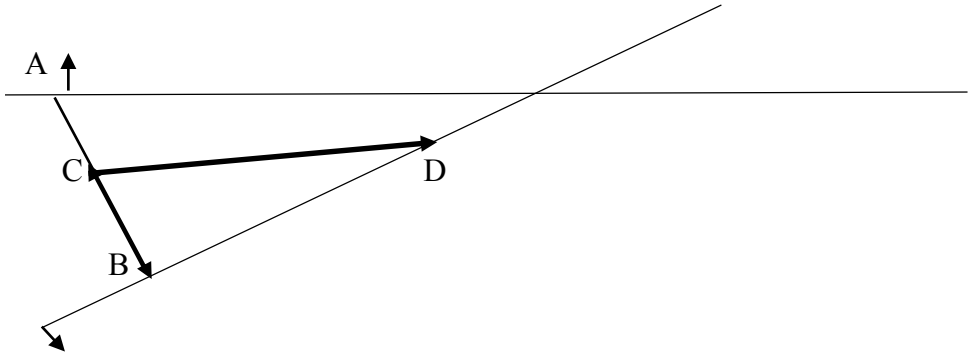


Figure 5.6 Half-step back and further expansion

A relaxation always consists of two parts: backward movement and expansion of the movement vector obtained from the basic CC method. In Figure 5.6 $\overrightarrow{AB}$ is the movement at the previous iteration having a tangent acceleration, $\overrightarrow{BC}$ is the backward movement, and $\overrightarrow{CD}$ is the expanded movement thereafter. Each part of the relaxation ($\overrightarrow{BC}$ and $\overrightarrow{CD}$) is executed at the different iterations. The simplest backward movement is a half-step back.

For high-dimensional problems, backward steps causing violation of additional constraints are explained below. Also, all violated constraints may not have equivalent feasibility distances, resulting in inefficient direction of the movement vector. However, it is expected that the backward step will partially improve this direction by reducing the variability between their feasibility distances. The optimal way to implement this task is very complicated, and approximations are used instead. The relaxation task is applied frequently and at some iterations it can show significant reduction of the feasibility distances after expansion.

The relaxation task is executed around every four iterations. An expansion step called after the backward movement increases the length of the movement vector using the ratios of the feasibility distances before and a short movement. As a result, it is simpler and faster than the tangent acceleration.

For expressions and transformations used in the following sections, the definition of the feasibility distance is extended. Negative values obtained from (2.2) are considered, and the following cases can occur:

- $fd_i > \alpha$: constraint *i* is violated. It is located on the half-plane which is not a part of a feasible region.
- $0 \leq fd_i \leq \alpha$: constraint *i* is satisfied because its violation is within the tolerance.
- $fd_i < 0$: constraint *i* is satisfied and located on the satisfied half-plane.

This slight abuse of nomenclature is needed for studying when a point satisfying a constraint is moved towards the constraint and violates it after crossing it.

5.7.1 Partial Step Back

The direction of backward movement is opposite to the previous movement vector $\vec{t}$. The size of the step back is based on the feasibility distances of selected constraints before and after the last movement. They are called the current and previous feasibility distances. Figure 5.7 shows examples of different categories of constraints after the step preceding the backward movement. Assume that the last movement vector is $\overrightarrow{OP}$, and some constraints are still violated. The following four groups of constraints are identified based on the points *O* and *P*:

- ***A*** is a group of constraints violated before and after the last movement, which is directed towards these constraints, but did not satisfy them due to the short length of vector $\overrightarrow{OP}$.
- ***B*** are the previously satisfied constraints which are violated after the movement.
- ***C*** are the previously violated constraints which are satisfied after the movement.
- ***D*** are constraints violated both before and after the last movement, which is directed opposite to these constraints, so their feasibility distances increase after the movement.

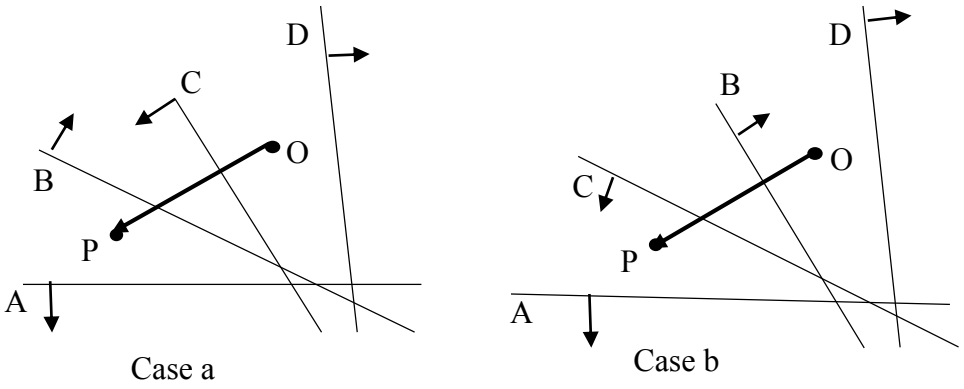


Figure 5.7 Two examples of states before backward movement

Some of these groups of constraints may not be present at the moment, and any group can contain multiple constraints. The constraint *C* in Figure 5.7 is the closest to the point P among all constraints from the group ***C*** if they exist. Similarly, the constraint *B* is the most distant from point P among all constraints from group ***B***. The following statements hold:

- At least one constraint from groups ***A***, ***B*** or ***D*** should be present. Otherwise, point P is already feasible, and the next step should not be carried out.
- At least one constraint from groups ***A*** or ***C*** should be present. Otherwise, the direction of the movement vector $\overrightarrow{OP}$ was incorrect.

The simplest case when groups ***A*** and ***D*** are empty is shown in Figure 5.8. As was mentioned above, the group ***B*** in this case is not empty.

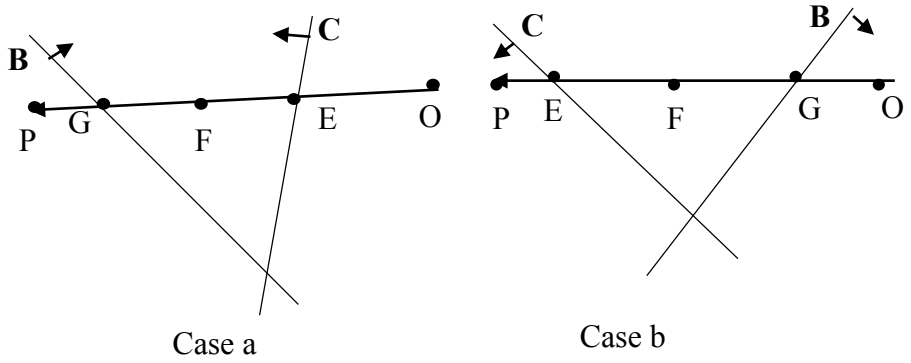


Figure 5.8 Groups B and C of violated constraints

In both cases in Figure 5.8, the movement vector passes over multiple constraints, and the status of one or more constraints from group ***C*** is changed from violated to satisfied, and the status of one or more constraints form group ***B*** is changed in the opposite manner. In both cases in Figure 5.8, $\overrightarrow{OP}$ was the last movement vector, and the points E and G are its intersections with the last passed constraint from the group ***C***, and the first passed constraint from the group ***B*** respectively. The point F located between E and G is the desired point after the backward movement: in case (a) it is feasible, and in case (b) all constraints from both groups are violated. As was shown above, the next movement vector can be directed to a feasible region (downward in case (b) of Figure 5.8). The algorithm uses values of the current feasibility distances fd_i^1 and the previous feasibility distances fd_i^0 for constraint *i*. They have the following properties:

- $\forall i \in \boldsymbol{C}$: $fd_i^1 < \alpha$ $fd_i^0 > \alpha$
- $\forall i \in \boldsymbol{B}$: $fd_i^1 > \alpha$ $fd_i^0 < \alpha$

Figure 5.9 shows relationships used in further transformations:

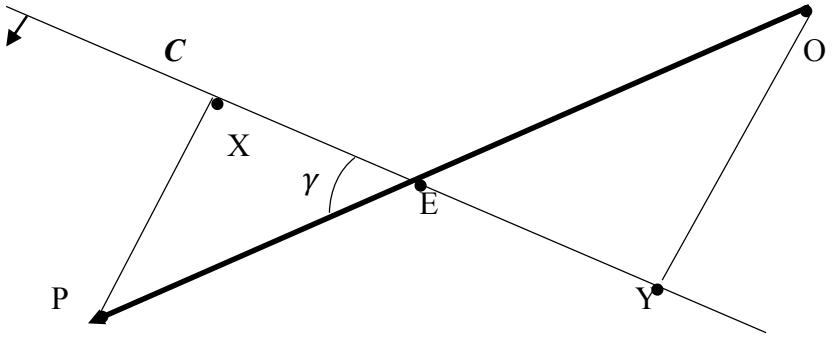


Figure 5.9 Relationships of vector lengths for constraint *C*

Let $\overrightarrow{t_0} = \overrightarrow{OP}$ be the last movement vector crossing a single constraint *C* at point E. Note that γ is the angle between the movement vector and the crossed constraint, and $q = {}^{1}/_{\sin\gamma}$. If |PC| = $|fd_i^1|$, then |PE| = q$|fd_i^1|$. Similarly, |OY| = $|fd_i^0|$ and |OE| = q$|fd_i^0|$.

Let $\overrightarrow{t_0}$ be the previous movement vector. Then, for any $i \in \boldsymbol{C}$, the backward movement vector due to constraint *C* is:

$$\overrightarrow{t_1} = \overrightarrow{PE} = \overrightarrow{PO} + \overrightarrow{OE} = -\overrightarrow{t_0} + \overrightarrow{t_0}\,\frac{|OE|}{|OP|} = \overrightarrow{t_0}\left(-1 + \frac{|OE|}{|OE|+|EP|}\right) = \overrightarrow{t_0}\left(-1 + \frac{q\times fd_i^0}{q\times fd_i^0 - q\times fd_i^1}\right) =$$

$$= \overrightarrow{t_0}\,\frac{-fd_i^0 + fd_i^1 + fd_i^0}{fd_i^0 - fd_i^1} = \overrightarrow{t_0}\left(\frac{fd_i^1}{fd_i^0 - fd_i^1}\right).$$

Here $\overrightarrow{t_0}$ is a movement vector at the previous iteration. The value of *q* does not need to be evaluated because it does not appear in the final expression. The possible values of the coefficient of $\overrightarrow{t_0}$ in the final expression lie in the interval [-1, 0] that agrees with the direction and the length of the backward movement.

If cardinality of the group $\boldsymbol{C}$ exceeds 1, the movement vector to the nearest constraint in this group is as follows:

$$\min_{i \in \boldsymbol{C}} |\overrightarrow{t_1}| = \max_{i \in \boldsymbol{C}}\left(\frac{fd_i^1}{fd_i^0 - fd_i^1}\right)\overrightarrow{t_0} \qquad (5.4)$$

Similarly, for $i \in \boldsymbol{B}$, the vector of backward movement to the constraint $\boldsymbol{B}$ is:

$$\overrightarrow{t_1} = \overrightarrow{PG} = \overrightarrow{PO} + \overrightarrow{OG} = -\overrightarrow{t_0} + \overrightarrow{t_0}\frac{|OG|}{|OP|} = \overrightarrow{t_0}(-1 + \frac{|OG|}{|OG|+|GP|} = \overrightarrow{t_0}(-1 + \frac{-q \times fd_i^0}{-q \times fd_i^0 + q \times fd_i^1}) =$$

$$= \overrightarrow{t_0}(-1 + \frac{fd_i^0 - fd_i^1 - fd_i^0}{fd_i^0 + fd_i^1}) = \overrightarrow{t_0}(\frac{-fd_i^1}{fd_i^1 - fd_i^0}).$$

The possible values of the coefficient by $\overrightarrow{t_0}$ in the final also lie in the interval [-1, 0]. If the cardinality of group $\boldsymbol{B}$ exceeds 1, the movement to the most distant constraint of this group is:

$$\max_{i \in \boldsymbol{B}} |\overrightarrow{t_1}| = \min_{i \in \boldsymbol{B}}(\frac{-fd_i^1}{fd_i^1 - fd_i^0})\, t_0 \qquad (5.5)$$

The movement to the desired point F is the average of the movements (5.4) and (5.5):

$$\vec{t} = 0.5\, (\max_{i \in \boldsymbol{C}}(\frac{fd_i^1}{fd_i^0 - fd_i^1}) + \min_{i \in \boldsymbol{B}}(\frac{-fd_i^1}{fd_i^1 - fd_i^0})\,)\, t_0 \qquad (5.6)$$

The more complicated case is when at least one of group $\boldsymbol{A}$ or $\boldsymbol{D}$ is not empty. The current and previous feasibility distances have the following properties:

- $\forall\, i \in \boldsymbol{A}$: $fd_i^0 > fd_i^1 > \alpha$
- $\forall\, i \in \boldsymbol{D}$: $fd_i^1 > fd_i^0 > \alpha$

The current feasibility distances to the constraints from the group $\boldsymbol{D}$, as well as the previous feasibility distances to the constraints from the group $\boldsymbol{A}$ can be relatively large, compared with those for groups $\boldsymbol{B}$ and $\boldsymbol{C}$. On the other hand, the vector $\overrightarrow{OP}$ can be almost parallel to one or more constraint from $\boldsymbol{A}$ or $\boldsymbol{D}$. As a result, the current and previous feasibility distances may not significantly differ, and an expression similar to (5.6) will have a very low value in its denominator, which will cause very large backward movements. The movement vector at the

resulting point may not point towards the feasible region, and as a result, some constraints may be highly violated. So these groups will only be checked for existence.

Table 5.1 contains all valid combinations of the four groups of constraints and descriptions of the corresponding backward movements. The main priority of the backward step is decreasing violations to Group ***D*** because they were increased by the last movement, and increasing violations to Group ***A*** because they were decreased by the last movement.

Table 5.1 Step back for combinations of the 4 groups of constraints

N	***A***	***B***	***C***	***D***	**Action**
1	0	> 0	> 0	0	Back between ***B*** and ***C*** (analyzed above)
2	0	> 0	> 0	> 0	Back between the closest ***C*** and previous point
3	0	0	> 0	> 0	Back between the closest ***C*** and previous point
4	> 0	0	0	0	No move back; continue to satisfy ***A***
5	> 0	0	0	> 0	Half-step back
6	> 0	0	> 0	0	No move back; continue to satisfy ***A***
7	> 0	0	> 0	> 0	Half-step back
8	> 0	> 0	0	0	No move back or back between the distant ***B*** and current point
9	> 0	> 0	0	> 0	Half-step back
10	> 0	> 0	> 0	0	No move back or back between the distant ***B*** and current point
11	> 0	> 0	> 0	> 0	Half-step back or back between ***B*** and ***C***
12	0	0	0	0	Impossible
13	0	0	> 0	0	Impossible
14	0	0	0	> 0	Impossible
15	0	> 0	0	0	Impossible
16	0	> 0	0	> 0	Impossible

All possible cases excepting the first which was analyzed above can be combined into the following groups:

- Group ***A*** is empty and group ***D*** is not (cases 2 and 3): these cases are shown in Figure 5.10. Only the constraint from group ***C*** which is closest to the point P is shown. The vector $\overrightarrow{OP}$ can pass over constraint *C* due to movement vector acceleration. Violations of

constraint(s) ***D*** are considered more important than those of group ***B*** (if they exist). Thus the step back should be to a point F located between points E and O. Its length is a partial case of (5.6) when the part corresponding to the group ***B*** is substituted by -1.

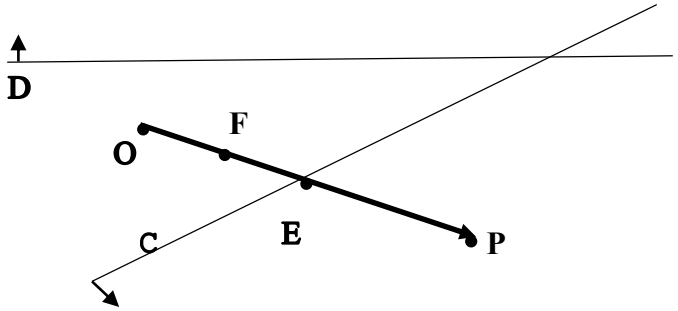


Figure 5.10 Group *D* exists and group *A* is empty

- Group A is not empty and group ***D*** is empty (cases 4, 6, 8): the backward movement is not needed because point P is already between ***A*** and ***B*** (if they exist) or all violated constraints lie on one side of the point P, and they were also violated at the previous iteration (no zigzag). When group ***B*** is not empty, another option is movement back to a point located between the points G and P. Its length is a partial case of (5.6) when the part corresponding to the group ***C*** is substituted by -1.

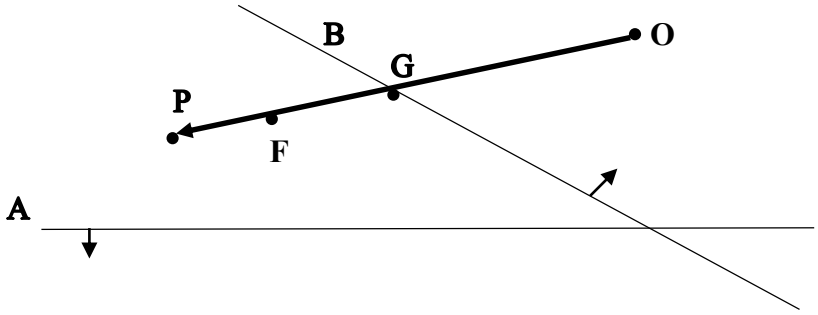


Figure 5.11 Group *A* exists and group *D* is empty

- Only groups ***A***, ***B***, ***C*** exist (case 10). In order to bring values of the existing violations ***A*** and ***B*** closer to each other, the target point F should lie between the points ***G*** and ***P***, as shown in Figure 5.11 (the group ***C*** is not shown because its presence does not change the backward movement):
- Both of groups ***A*** and ***D*** are not empty (cases 5, 7, 9, 11). Regardless of groups ***B*** and ***C***, the half-step back is needed. Due to the direction of the last movement, violations of group ***A*** are considered relatively low, whereas violations of group ***D*** are considered relatively high. The half-step back is expected to decrease their difference.

Based on the table 5.1, the length of backward movement can depend on the existence of group ***B*** only if group ***D*** is empty. Similarly, the length of the backward movement can depend on the existence of group ***C*** only if group ***A*** is empty.

The approaches explained above are simple and the corresponding algorithms are relatively quick. The backward movement may or may not improve the direction of the next movement, as is discussed in the conclusions.

It should be noted that the backward movement step starts as a common iteration with finding all feasibility distances, testing constraints for violation and updating the corresponding arrays for the current and previous iteration. A counter manages the frequency of applying the backward movement. Also, this step is postponed if there is only one violated constraint which is the same during the last two iterations. The backward step algorithm is shown in Algorithm 5.9.

Inputs:

- fd^1: *m* vector of feasibility distances for current iteration.
- fd^0: *m* vector of feasibility distances for previous iteration.
- $\boldsymbol{t}^0$: previous movement vector.
- tolerance *tol* = 1e-6

1. For each constraint *i* among all constraints violated at previous iteration:
   1.1. If the $fd_i^1 \leq tol$:
      1.1.2. $maxC \leftarrow \max(maxC, \frac{fd_i^1}{fd_i^0 - fd_i^1 + tol})$; continue the loop.
   1.2. If $fd_i^1 < fd_i^0$: group ***A*** exists.
   1.3. If $fd_i^1 > fd_i^0$: group ***D*** exists.
2. If group ***D*** is empty:
   2.1 For each constraint *i* among all currently violated constraints:
      2.1.1. If $fd_i^0 \leq tol$:
         2.1.1.1. $minB \leftarrow \min(minB, \frac{fd_i^1}{fd_i^0 - fd_i^1 - tol})$.
3. If group ***A*** is not empty:
   3.1. If group ***D*** is not empty:
      3.1.1. For each variable *j*:
         3.1.1.1. $x^j \leftarrow x^j - 0.5t_j^0$;
      3.1.2. Exit.
   3.2. If group ***B*** is not empty:
      3.2.1. For each variable *j*:
         3.2.1.1. $x_j \leftarrow x_j + 0.5minB \times t_j^0$;
   3.3. Exit.
4. If group ***D*** is not empty:
   4.1. For each variable *j*:
      4.1.1. $x_j \leftarrow x_j + 0.5(maxC - 1)t_j^0$;
   4.2. Exit.
5. For each variable *j*:
   5.1. $x_j \leftarrow x_j + 0.5\,(maxC + minB)\,t_j^0$

Algorithm 5.9 Backward movement step

5.7.2 Expansion

After a step back, the total violations at the updated point are higher, but its position may be better for expansion of the next movement vector. An iteration of an expansion task consists of the following steps:

- Applying the basic CC algorithm to obtain an initial movement vector and updating the point location.
- Analysis of direction of the movement vector and possibly increasing its length, called an expansion, and adjusting the point location.

As a result an iteration of this type can contain only one movement (rare case) or two movements (most probable case). The movement vector is started from a "better" point, with its direction obtained from the basic CC algorithm which contains contributions from all of the violated constraints.

Possibly increasing the length of the movement vector is based on an analysis of the feasibility distances of the constraints violated before any expansion. If the movement vector is directed towards a feasible region, the previous short movement vector should decrease feasibility distances for all violated constraints. The constraints will be satisfied or less violated. Then this movement vector can be expanded for additional decrease of feasibility distances of the less violated constraints.

Figure 5.12 shows relationships which are used in further transformations:

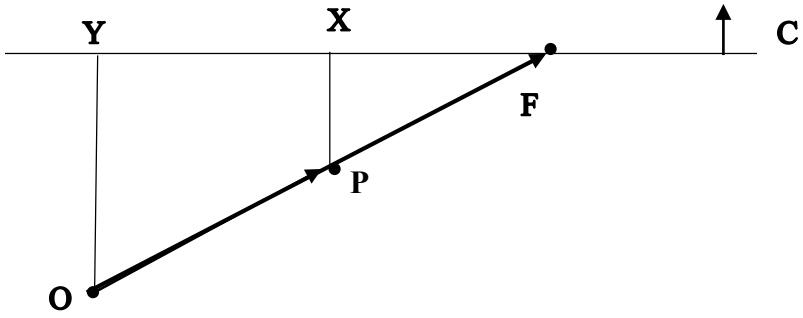


Figure 5.12 Relationships of vector lengths at expansion step

Let $\overrightarrow{OP}$ be the movement vector at the initial step. It decreases violation of the constraint $\boldsymbol{C}$ which is still violated, and can be expanded to satisfy it. Note that |OY| = fd^0 > |PX| = fd^1 > 0. If $\overrightarrow{PF}$ is an expanded vector eliminating violation of the constraint $\boldsymbol{C}$, then:

$\frac{fd_0}{fd_1} = \frac{|OF|}{|PF|} = \frac{|OP|+|PF|}{|PF|} = 1 + \frac{|OP|}{|PF|}$, and

$$|PF| = |OP|\frac{fd^1}{fd^0 - fd^1} \tag{5.7}$$

Expression 5.7 can be applied for all constraints violated after the initial step. Selecting the maximum coefficient at |OP| would satisfy all these constraints. However, this strategy cannot be applied because the movement vector may not be directed exactly towards a feasibility region. As a result, the following problems may occur:

1. Some of the violated constraints can become more violated after the initial step, and further expansion of the movement vector will increase the violations; this is easily identified after the initial step. When the increase is not large, they can reduced in later iterations. On the other hand, if the movement vector $\overrightarrow{OP}$ is almost parallel to a constraint $\boldsymbol{C}$, the value of the denominator in (5.7) is very small, which causes a very large expansion and enormous violation of such constraints.
2. The same effect is produced by the constraints which were satisfied before the initial step, and became violated after it. They are also easily identified after the initial step.
3. The algorithm does not analyze the constraints which were satisfied before and after the step 1, but became “less oversatisfied”. They can become highly violated after a large expansion, but this case cannot be identified after the initial step.

Consequently, finding the maximum value of the coefficient in (5.7) for all violated constraints should not be applied. The expansion task is expected to satisfy some of the violated constraints, and not to produce other highly-violated constraints.

After the initial step, the changes in feasibility distances of all constraints violated before the step are evaluated. If all of them become satisfied, which is rare due to the shortness of the movement vector, the task is finished. The next iteration will check if the point is feasible or whether other constraints are newly violated. Otherwise, two cases may occur:

- All constraints became less violated, and an expansion is desired.
- Some of constraints became more violated, and a possible expansion should not be large or should be omitted. This case occurs due to an ineffective movement direction.

In order to correctly determine the case, only the constraints that are still violated after the initial step are considered. The ratio of their feasibility distances before and after the first step is calculated. Let $r^i = {fd_i^0}/{fd_i^1}$. A group of constraints having ratios *r* above 0.99 is identified. It contains the constraints with decreased violations (when the ratio exceeds 1) and the constraints with insignificant increased violations (when the ratio is slightly below 1). If this group is empty, an expansion should be undertaken, because all violated constraints become significantly more violated. On the other hand, if all ratios exceed 0.99, an expansion can be effective. Using the maximum value in (5.7) for multiple problems showed significant worsening of results due to problem 3 described above. Using a lower expansion ratio will not produce enormous violations for non-controlled constraints. For this case, the expansion ratio is calculated as an average of the coefficients (5.7) for all the constraints violated before the initial step that became less violated thereafter.

The last case is when not all the ratios exceed the value of 0.99. In this case violation of some constraints can be significantly increased after expansion. Expansion is omitted if after the first step at least 10% of previously violated constraints became satisfied, because this percentage is considered good performance of a step. Expansion is also omitted if at least one ratio r has a value below 0.5, because it will cause a significant increase of violation of the corresponding constraint. Otherwise, an average ratio of all constraints violated before the step 1 is calculated. If its value is below 1, many constraints will be worsened after an expansion, and it is omitted. If its value exceeds 1, it is used for evaluating the expansion coefficient in (5.7), but the length of the expanded vector is not allowed to exceed one at the first step.

The expansion algorithm explained above is shown in Algorithm 5.10. Note that all constants, such as minimum ratio coefficients or minimum number of satisfied constraints are evaluated and adjusted after multiple runs on various problems. The effect of changing their values is unknown.

It is not expected that an expansion will significantly decrease the worst violation, but some expansions may do so.

Inputs:

- $\boldsymbol{fd^0}$ *m* vector of feasibility distances after the last iteration.
- ***viol*** indexes of violated constraints after the last iteration.
- n_{viol}: number of violated constraints after the last iteration.
- Tolerance *tol* = 1e-6

1. For each variable *j*:
   1.1. Find movement vector $\boldsymbol{t}^0$ from basic CC algorithm and update $\boldsymbol{x}$
2. For each constraint *i* in ***viol***:
   2.1. Find updated feasibility distance fd_i^1
3. Find number of constraints n_{vv} violated before and after Step 1.1
4. If n_{vv} = 0: exit
5. For each constraint *i* violated before and after Step 1:
   5.1. $r_i \leftarrow \frac{fd_i^0}{fd_i^1}$
6. Find number of improved constraints n_{impr} with $r_i > 1 + tol$ and number of constraints n_{1impr} not significantly worsened with $r_i > 0.99$
7. If $n_{1impr} = 0$: exit
8. If $n_{impr} = n_{vv}$
   8.1. $meancross \leftarrow 0.0$; $n_{impr} \leftarrow 0$
   8.2. For each constraint *i* violated before and after Step 1.1:
      8.2.1. If $r_i > 1$:
         8.2.1.1. $meancross \leftarrow meancross + \frac{1}{r_i - 1}$
         8.2.1.2. $n_{impr} \leftarrow n_{impr} + 1$
   8.3. $meancross \leftarrow meancross / n_{impr}$
   8.4. For each variable *j*:
      8.4.1. $x_j \leftarrow x_j + meancross \times t_j^0$
   8.5. Exit
9. If $n_{viol} - n_{vv} > 0.1 \times n_{viol}$: exit
10. If $\min(r_i) < 0.5$: exit.
11. $ar \leftarrow$ average ratio of r_i for all constraints violated before and after Step 1
12. If $ar < 1$: exit
13. If $ar > 2$: $ar \leftarrow 2$
14. For each variable *j*:
    14.1. $x_j \leftarrow x_j + ar \times t_j^0$

Algorithm 5.10 Movement vector expansion

6. Concurrent Computation of New Algorithms

6.1 Introduction

As mentioned in Section 2.2.4, the new CC algorithms, like the most projection algorithms, can be computed in parallel or concurrent configuration. Implementation on a massively parallel computing platform, such as CUDA can create inefficient code, because the new CC algorithms make extensive use of pointers and conditional statements. In addition, these algorithms use sparse matrix structures containing only non-zero values, and the number of these values can be significantly different for various rows. As a result, nodes executed in parallel can have various volumes of computations that reduces the speedup.

This research is based on concurrent computing which can be implemented on multiple cores of a computer. Various cores can execute different code on different data. One of the simplest configurations for concurrent computing, which is used in this research, is the following:

- For p cores, generate p different initial points.
- Each core executes the same or a different CC algorithm starting from the different initial points. Each run can be restricted by a maximum number of iterations or by a maximum execution time.
- If one of cores successfully finds a feasible point before reaching a maximum restriction, it produces its output and sends messages to the other cores to stop their execution, which is no longer needed. If none of them finishes earlier, the result provided by any of the cores can be used.

In this scenario, the common model parameters are stored in shared memory with independent access for all cores. In addition, each core uses its own small amount of memory for local variables

and data. All cores run independently, and pass messages only at the end, which reduces message passing time. In more complex scenarios, a core may be reset to a different algorithm or initial point before completing its task, if the current temporary result is very poor.

Let t_i be the algorithm computation time for core i (among p cores). Then the time for concurrent computing $T(p)$ (assuming that the communication time is very low) is a random value equal to $\min_i t_i$, i.e. the computation time of the core that first finds a feasible point. The time for sequential computing $T(1)$ is also a random value. Consider that a single core can select any initial point with equal probability. Then, the expected value of $T(1)$ is equal to $\frac{\sum_{i=1}^{p} t_i}{p} = \underset{i}{mean}(t_i)$.

Consider two extreme cases for this scenario. The first case is when the computing times for various initial points are significantly different, e.g. $t_1 = 1; t_2 = 5$. In this case $T(p) = 1$ and $T(1) = 0.5(1 + 5) = 3$. The speedup is then $s(p) = T(1) / T(p) = 3 > p = 2$, and for randomized algorithms we can have super-linear expected speedup which is theoretically impossible for non-randomized concurrent algorithms. In the second case all computing times are similar, e.g. $p = 9; t_i = 0.99 + 0.01 * i$. In this case $T(p) = 1$ and $T(1) = \frac{\sum_{i=1}^{9}(0.99+0.01i)}{9} = 1.04 \ll p$. In this case, the concurrent implementation is very inefficient, and it is better to use a single initial point for all cores with parallel matrix and vector computations. It should be noted that the second case with similar computation times for various initial points is not expected for real problems.

These examples show the impact of larger variability of solution times for different algorithms and initial points. The results are unknown at the start, and one of the ways to increase the variability of solution times is by selecting different initial points, which are “far” from each other.

6.2 Minimum Set of Algorithms

The total number of algorithms produced by combining the various tangent acceleration methods and settings defined in sections 5.6.2 – 5.6.4 is very large. They can have different efficiencies for various problems and initial points. Some of the algorithms using extremely accelerated movement vectors may have rare very good performance and high frequency of failure. Other algorithms rarely fail, but their speed is never very good. Thus the following should be considered in concurrent implementation of the new CC methods:

1. If the number of cores is not very large, then not all of the algorithms can be concurrently executed. The number of algorithms used should be small. Finding the best minimum set of algorithms for a non-large number of cores is an optimization problem, which is analyzed in this section.
2. The set of selected algorithms should have high concurrent performance, which depends on the "best" sequential algorithm in this set. The main performance measurement of a sequential algorithm is its computation time. A simpler metric mentioned in many publications is the number of iterations, which is used at this step of this research. More generally, performance of an algorithm can be measured as follows:
    - If successful: the number of iterations needed to satisfy the distance tolerance.
    - If unsuccessful: if the distance tolerance is not satisfied within the maximum number of iterations, then the smallest maximum feasibility distance over all iterations.

Simple selection of algorithms with a higher average performance is not applicable as shown in Table 6.1, which contains an artificial example of the number of iterations needed to solve 4 problems by each of 5 algorithms. The first three algorithms are more efficient because they have a lower average number of iterations. However, the algorithms having the worst average

performance (4 and 5) can be used concurrently, and are expected to produce the fastest result among other pairs of algorithms.

Table 6.1 Comparison of 5 Algorithms Over 4 Problems (number of iterations)

	Problem 1	Problem 2	Problem 3	Problem 4	Total iterations
Algorithm 1	20	100	100	20	240
Algorithm 2	20	20	100	100	240
Algorithm 3	20	100	20	100	240
Algorithm 4	1000	20	20	1000	2040
Algorithm 5	20	1000	1000	20	2040

6.2.1 Optimization Model for Selecting the Best Set of Algorithms

Comparison of algorithms is based on their relative performance. If all algorithms are applied to the same problem with the same initial point, the vector of their outputs consists of the minimum number of iterations to finish (in the case of success) or the lowest value of the maximum feasibility distance (in the case of failure). Then, the result of a set of algorithms *s* is as follows (where *niter* is the number of iterations):

- If at least one algorithm succeeds: $\min_{s} niter$
- Else (all failed): $\min_{s} (\max_{i \in m} fd_i)$

The outputs of the algorithms are collected into two groups. One algorithm is given the score 1 and assigned to the "best" group, and one algorithm is given the score 0 and assigned to the "worst" group. All outputs from the best group are deemed acceptable, whereas all outputs from the worst group are deemed unacceptable. The following separating rule is used:

- If at least one algorithm succeeds, the algorithm succeeding with the fewest iterations has score 1. All other algorithms that succeed with fewer than twice the number of

iterations as the fastest one are also assigned score 1. Other successful algorithms having more iterations, and those that fail are given the score 0.

- If all algorithms fail, the algorithm with lowest maximum feasibility distance is given the score 1. All algorithms having a maximum feasibility distance less than 50% more than the lowest one, also given the score 1. All other algorithms are given the score 0.

The resulting vector of algorithm scores always has at least one algorithm with score 1, even if no algorithm succeeded. If the results have low variability, then the number of algorithms with score 1 may be as high as the total number of the algorithms.

The process of selecting the best sets of algorithms is as follows:

- Each algorithm S_i, (i = 1, 2, …, I) is applied for all combinations of problems D_j (j = 1, 2, …, J) and initial points P_k (k = 1, 2, …, K). In the resulting matrix, the number of rows equals the number of algorithms, and the number of columns equals the product of the number of problems J and the number of random initial points K.
- For each combination of a problem and an initial point: the best result among all algorithms is selected and the performances of all algorithms are scored as 1 or 0 as described above.
- Using the $I \times KJ$ matrix of scores V, the best minimum set of algorithms is determined.

The best minimum set of algorithms is defined as follows:

- It should contain the minimum possible number of algorithms subject to the condition below.

- Being applied concurrently to any problem, the best minimum set of algorithms produces acceptable outputs for at least *qp* initial points among *K*. For a single initial point, an acceptable output can be produced by one or more algorithms from the best minimum set.

6.2.2 Binary Linear Programming Model

Let $\boldsymbol{X} = [x_1, \dots, x_I]^T$ be a binary vector with the possible values

$$x_i = \begin{cases} 1, & \text{if algorithm } i \text{ is present in a set of algorithms} \\ 0, & \text{otherwise} \end{cases}$$

The best minimum set of algorithms is described by the vector $\boldsymbol{X^{opt}}$ indicating the algorithms included/excluded in the best minimum set. Its values are determined by integer linear programming.

For a selected problem *j* and initial point *k* an expression $V_{.jk}X$ can have the following possible values:

$$V_{.jk}X = \begin{cases} \geq 1, & \text{if a set of algorithms produces at least one acceptable output for problem } j \text{ and initial point } k \\ 0, & \text{all outputs produced by the set of algorithms for problem } j \text{ and initial point } k \text{ are unacceptable} \end{cases}$$

To permit occasional unacceptable outputs when applying the minimum set of algorithms, we introduce binary artificial variables e_{jk} for each problem *j* and initial point *k.* Then a vector of variables $[x_i \; e_{jk}]$ contains *I* decision variables *x* and *J* * *K* elastic variables *e*. The first set of constraints specifies that both the decision and elastic variables contribute to a successful result for each combination of a problem and an initial point:

for each *j* and *k*: $\sum_{i=1}^{I} V_{ijk} * X_i + e_{jk} \geq 1$; total *J* * *K* constraints (6.1)

The desired case is when the elastic variables $e_{jk} = 0$, which happens when the set of algorithms produces at least one acceptable output for all problems and all initial points. When some elastic variables are equal to 1, all algorithms produce unacceptable outputs for the corresponding problem and initial point. The sum of the non-zero elastic variables shows the count of unacceptable outputs. The following set of *J* constraints defines the maximum number of unacceptable outputs for each problem *j*:

$\sum_{k=1}^{K} e_{jk} \leq K - qp$ for each *j*, total *J* constraints (6.2)

The objective is to minimize the number of algorithms in the optimal set as follows:

Minimize: $\sum_{i=1}^{I} X_i$ (6.3)

This binary linear model can be solved by any MILP solver. However multiple optimum solutions satisfying (6.1), (6.2) and (6.3) are possible. These multiple optimum solutions correspond to different minimum sets of algorithms having the same cardinality but different members. They can have different proportions of acceptable outputs, when all proportions exceed the minimum stated in Eqn. 6.2. For example, if each of 2 problems should be solved for at least 6 starting points among 10, the following sums of elastic variables for 3 optimal cases do not violate constraint 6.2, as shown in Table 6.2:

Table 6.2 Comparison of optimal solutions

	D_1	D_2
$\sum_{k=1}^{K} e_{jk}$ for optimal solution 1	4	3
$\sum_{k=1}^{K} e_{jk}$ for optimal solution 2	2	4
$\sum_{k=1}^{K} e_{jk}$ for optimal solution 3	3	2

However optimum solution 3 is preferable because it has acceptable outputs for more starting points. Thus, the modified optimal solution can be found as follows:

- Find the minimum number of algorithms satisfying (6.1) and (6.2).
- For all minimum sets of algorithms, select the one with the highest efficiency.

All possible optimum solutions have the same number of algorithms n^s, which is used in the second stage. It has the same set of constraints (6.1) and (6.2). An additional constraint uses the previously obtained minimum size of the set as follows:

$$\sum_{i=1}^{I} X_i = n^s \tag{6.4}$$

Any solution satisfying constraints (6.1), (6.2), and (6.4) is optimum. The best of them should have the minimum number of penalized cases, as expressed by the second stage objective function:

$$\text{Minimize } \sum_{k=1}^{J*K} e_{jk} \tag{6.5}$$

Multiple solutions defined by (6.4) and (6.5) can also occur. All of them will have the same minimum cardinality of the set of algorithms, as well as the same maximum total proportion of acceptable outputs, and are thus equivalent. Any of them can be taken for further consideration.

The initial optimization model satisfying (6.1), (6.2) and (6.3) can be omitted if the number of algorithms is known, or the set of algorithms should not necessarily have the lowest cardinality. For example, if the number of cores is known and all cores compute different algorithms.

6.3 Expected Performance of Concurrent Computing on a Small Number of Cores

The expected performance of concurrent computing can be analytically determined from outcomes obtained from sequential computing. Consider the scenario where *M* cases, which are combinations of different algorithms and initial points are computed sequentially. The

performance outcomes x_i , (i = 1 : M) are sorted such that x_1 is the best outcome (the fastest result or the lowest maximum feasibility distance), and x_M is the worst outcome. These sorted results can be used to calculate the expected performance of concurrent computing of these cases assuming that the concurrent computing is performed without messaging delays.

If all the cases are computed on $p = M$ cores, the expected outcome is equal to x_1, which is obtained by the core producing the fastest result. On the other hand, for a single core (p = 1), only one case among M can be computed. Each case is selected randomly with the same probability $1/M$. Then the expected outcome is $\frac{1}{M}\sum_{i=1}^{M} x_i = mean(x)$.

Consider the more complicated cases when $1 < p < M$. If $p = M - 1$, the best case x_1 can be selected for one of the $M - 1$ cores with probability $\frac{M-1}{M}$. The probability that the best case is not selected for any core is $1/M$. Then, the second best outcome will be selected for one of cores, and the expected outcome is $\frac{M-1}{M}x_1 + \frac{1}{M}x_2$.

Similarly, for $p = M - 2$, the expected outcome is $\frac{M-2}{M}x_1 + \frac{2}{M}(\frac{M-2}{M-1}x_2 + \frac{1}{M-1}x_3)$.

More generally, for $p = M - J$, the expected outcome is as follows:

$$\frac{M-J}{M}x_1 + \frac{J}{M}(\frac{M-J}{M-1}x_2 + \frac{J-1}{M-1}(\frac{M-J}{M-2}x_3 + \frac{J-2}{M-2}(\dots)\dots)$$

Therefore, the expected outcome of concurrent computing of M cases on $1 \le p \le M$ cores is as follows:

$$x_{exp} = \sum_{i=1}^{J+1} c_{iJ}x_i \text{ , where } J = M - p,\ M \in [0,\ M-1]; \tag{6.6}$$

$$c_{iJ} = \frac{M-J}{M-i+1}\prod_{k=1}^{i-1}\frac{J-k+1}{M-k+1},\ i \le J+1 \tag{6.7}$$

7. Experimental Setup

7.1 Introduction

Two sets of experiments were performed:

- Algorithm tuning. The best CC algorithms and parameter settings are selected using a training set problems.
- Algorithm testing. The performance of the tuned CC algorithms is evaluated on a testing set of problems.

7.2 Metrics of Algorithm Performance

Each algorithm has a maximum 5000 iterations to find a feasible point. If one is found, i.e. feasibility distances for each constraint and bound are below the tolerance α equal to 10^{-5}, the number of iterations is the performance measurement. All three values introduced in Section 2 (maximum feasibility distance, sum of feasibility distances, and number of violated constraints) are stored, but the maximum feasibility distance is the main metric of interest. All of these values tend to decrease as the algorithms iterate, but the decrease is not monotonic, and the last iteration may not have the lowest value of the maximum feasibility distance. The best value of the maximum feasibility distance seen at any iteration is saved. If the output solution violates the maximum feasibility distance tolerance, then the performance is evaluated as follows:

$$\min_{iter} \max_{i} fd_i \tag{6.8}$$

The values of the other two metrics are noted at the iteration having the lowest value of maximum feasibility distance, and hence may not have the lowest value seen among all

iterations. The computing time is also stored, but its value depends directly on the number of iterations.

7.3 Algorithms Compared

7.3.1 New Algorithms

The new CC algorithms described in Section 5 (basic CC algorithm, maximum unique direction, tangent unique direction and generalized direction algorithms) consist of various acceleration tasks and common step-back-and-expansion. In addition, they have different settings defined in Section 5.6, and can optionally use Algorithm 5.7 for flexible values of α. Common to all algorithms is the feasibility presolving implemented in this research. The presolving tasks are: removal of singleton rows, removal of fixed variables, handling of uniform-sign columns, and constraint tightening.

7.3.1.1 Increasing the Length of a Feasibility Vector

Options for increasing length of the feasibility vector are described in Section 5.6.2. Table 7.1 summarizes these options for the various CC methods. It contains multipliers of the vector length for various types of dimensions. For each algorithm, option 1 corresponds to the version without additional increasing of the length of feasibility vectors.

Table 7.1 Options for increasing length of a feasible vector

Method	Option	Unique sign dimensions	Other dimensions
1. BA	1	$maxtan^{1.0}$	
	2	$maxtan^{1.2}$	
	3	$maxtan^{1.4}$	
2. TUDA	1	$meantan$	1
	2	$meantan^{2}$	$meantan$
	3	$meantan^{1.4}$	$meantan^{0.7}$
3. MAUD	1	t^{max}	1
	2	t^{max}	$meantan$
	3	$t^{max} \times meantan$	$meantan$

4. GDA	1	$maxtan$	1
	2	$maxtan^{2}$	$maxtan$
	3	$maxtan^{1.4}$	$maxtan^{0.7}$
	4	$maxtan^{2.4}$	$maxtan^{1.2}$

In addition, some tests use an additional version of the CC algorithm, which has only relaxation (step back and expansion) without tangent acceleration. It is labelled Method 0. As a result, the total number of algorithm variants is 14.

7.3.1.2. Additional Improvement of the Movement Vector

This option is described in Section 5.6.3. The extremal case $pf = 0$ gives poor results, and is omitted from the possible settings. This research considers values of pf equal to 1.0, 0.8 and 0.5. An additional option is using $pf = 0.5$ only for constraints which are parts of ill-conditioned pairs and $pf = 1$ for other constraints. There are 4 total different settings of this type.

7.3.1.3 Improving Selected Violated Constraints

This setting is described in Section 5.6.4. There are 3 options of this type. The first one does not use Algorithm 5.8; the second one applies this algorithm with modified Step 2.4 leading to the largest multiplication; the third one is obtained by applying the non-modified algorithm.

The total number of new algorithm variants obtained by all combinations of the settings listed in Sections 7.3.1, 7.3.2 and 7.3.3 is $14 \times 4 \times 3 = 168$.

7.3.2 Existing Algorithms

The existing CC algorithms selected for comparison were developed by Ibrahim and Chinneck [17]:

- Basic CC algorithm described in Algorithm 2.1.

- Feasibility-Distance Based (*FDfar*) CC algorithm which sets a higher priority to the most distant constraint.
- Average-Direction Based (*DBavg*) CC algorithm selects the direction of movement for each dimension depending on the number of positive and negative components of the feasibility vectors.
- Maximum Direction-Based (*DBmax*) CC algorithm selects the direction of movement for each dimension depending on the largest positive or negative movement.

The partial presolving described in Section 7.3.1 is also applied for the exisiting algorithms

7.4 Software and Hardware Environment

The preparation and experimental setup tasks were implemented on *R* [25]. The tasks include:

- Extracting Netlib files having a MPS format using the *Rglpk_read_file* function from the *Rglpk R* package [25].
- Manipulation of the sparse matrices for transforming the problem to the standard form ($\leq$ constraints).
- Creating sparse structures for tangent acceleration and the *GDA* algorithm.
- Implementing linear optimization for finding the best minimum set of new algorithms.
- Analysis of experimental results.

All the new and existing CC algorithms were implemented in C. Inputs to the C programs are sparse structures produced by R, and the list of algorithms to execute (all 168 algorithms when finding the best minimum set, and selected algorithms and their settings for various tests). The C programs produce the number of iterations, computing time, values of the obtained vector $\boldsymbol{x}$, the

lowest value of maximum feasibility distance and the corresponding number of violated constraints and their total feasibility distance. The C programs were compiled and executed using the Code Block IDE (version wx2.8.12) [14].

All algorithms are implemented for sequential computing. The estimated concurrent computing results are estimated from the sequential computing results.

Both R scripts and C programs were run on Windows 8.1 and 2-core ARM AMD8-6500 3.5 GHz processor with 6 GB of RAM.

7.5 Model Selection

All models are taken from the Netlib repository of linear programming models [22]. Some very large-size problems could not be tested due to memory problems in the current version of the R script for loading files.

86 Netlib problems are used in total. They are randomly split onto two disjoint sets: the training set (40 problems) and the testing set (46 problems). Descriptive statistics of the full set of problems are given in Table 7.2. The training set problems are in bold.

Table 7.2 Descriptive statistics of 86 problems

Problem	Number of non-zero coefficients	Original number of constraints	Number of variables	Number of equality constraints	Number of finite bounds	Total number of constraints
25fv47	10400	821	1571	516	1571	2908
80bau3b	21002	2262	9799	0	9799	12061
adlittle	383	56	97	15	97	168
afiro	83	27	32	8	32	67
agg	2410	488	163	36	163	687
bandm	2494	305	472	305	472	1082
beaconfd	3375	173	262	140	262	575
blend	491	74	83	43	83	200
bnl1	5121	643	1175	232	1175	2050
bnl2	13999	2324	3489	1327	3489	7140

boeing1	3819	440	384	9	384	833
boeing2	1283	185	143	4	143	332
bore3d	1429	233	315	214	315	762
brandy	2148	220	249	166	249	635
CAPRI	1767	271	353	142	339	752
cre-a	14987	3516	4067	335	4067	7918
cre-c	13244	3068	3678	335	3678	7081
CYCLE	20720	1903	2857	1389	2850	6142
CZPROB	10669	929	3523	890	3523	5342
D2Q06C	32417	2171	5167	1507	5167	8845
D6CUBE	37704	415	6184	415	6184	7014
degen2	3978	444	534	221	534	1199
degen3	24646	1503	1818	717	1818	4038
E226	2578	223	282	33	282	538
FF800	6227	524	854	350	854	1728
FINNIS	2310	497	614	47	614	1158
FIT1D	13404	24	1026	1	1026	1051
FIT1P	9868	627	1677	627	1677	2931
GANGES	6912	1309	1681	1284	1681	4274
GFRD-PNC	2377	616	1092	548	1092	2256
GROW7	2612	140	301	140	301	581
GROW15	5620	300	645	300	645	1245
GROW22	8252	440	946	440	946	1826
ISRAEL	2269	174	142	0	142	316
KB2	286	43	41	16	41	100
ken-07	8404	2426	3602	2426	3602	8454
lotfi	1078	153	308	95	308	556
MAROS	9614	846	1443	323	1443	2612
modszk1	3168	687	1620	687	1618	2992
nesm	13680	750	2923	480	2923	4153
PEROLD	6018	625	1376	495	1288	2408
pilot	43167	1441	3652	233	3652	5326
PILOT4	5141	410	1000	287	912	1609
PILOT-JA	14698	940	1988	661	1900	3501
PILOTNOV	13057	975	2172	701	2172	3848
PILOT-WE	9126	722	2789	583	2709	4014
QAP8	7296	912	1632	912	1632	3456
RECIPELP	663	91	180	67	180	338
SC50A	130	50	48	20	48	118
SC50B	118	50	48	20	48	118
SC105	280	105	103	45	103	253
SC205	551	205	203	91	203	499
SCAGR7	420	129	140	84	140	353
SCAGR25	1554	471	500	300	500	1271
SCFXM1	2589	330	457	187	457	974

SCFXM2	5183	660	914	374	914	1948
scfxm3	7777	990	1371	561	1371	2922
SCORPION	1426	388	358	280	358	1026
SCRS8	3182	490	1169	384	1169	2043
SCSD1	2388	77	760	77	760	914
SCSD6	4316	147	1350	147	1350	1644
SCSD8	8584	397	2750	397	2750	3544
SCTAP1	1692	300	480	120	480	900
SCTAP2	6714	1090	1880	470	1880	3440
SCTAP3	8874	1480	2480	620	2480	4580
SEBA	4367	522	1028	507	1028	2057
SHARE1B	1151	117	225	89	225	431
SHARE2B	694	96	79	13	79	188
SHELL	3556	536	1775	534	1775	2845
SHIP04L	6332	402	2118	354	2118	2874
SHIP04S	4352	402	1458	354	1458	2214
SHIP08L	12802	778	4283	698	4283	5759
SHIP08S	7114	778	2387	698	2387	3863
SHIP12L	16170	1151	5427	1045	5427	7623
SHIP12S	8178	1151	2763	1045	2763	4959
SIERRA	7302	1227	2036	528	2036	3791
STAIR	3856	356	467	209	461	1026
STANDATA	3031	359	1075	160	1075	1594
STANDGUB	3139	361	1184	162	1184	1707
STANDMPS	3679	467	1075	268	1075	1810
STOCFOR1	447	117	111	63	111	291
STOCFOR2	8343	2157	2031	1143	2031	5331
TUFF	4520	333	587	292	585	1210
VTP-BASE	908	198	203	55	202	455
WOOD1P	70215	244	2594	243	2594	3081
WOODW	37474	1098	8405	1085	8405	10588
Average	7948.2	676.7	1577.0	397.1	1572.7	2646.5

8. Algorithm tuning

8.1 Pairwise Comparisons

There are 3 pairwise comparisons: (i) flexible vs. fixed feasibility distance tolerance, (ii) presence vs. absence of the tangent acceleration, and (iii) presence vs. absence of step back. Each compared pair differs only by the specified setting; all other settings and initial points are the same. The best between the two options is selected as follows:

- If both of algorithms exceed the maximum number of iterations: algorithm with lower best maximum feasibility distance.
- Otherwise: the algorithm that finishes earlier (lower number of iterations).

The first option is better than the second it is assigned a score of 1, and a score of -1 otherwise. The comparisons are repeated for all combination of the other settings. The total sum of the scores is divided by number of comparisons. As a result, an average score of 1 indicates that the first option performs better in all cases, and -1 conversely. An average score 0 indicates that both options are equivalent.

8.1.1 Feasibility Distance Tolerance

This test compares the performance of flexible versus fixed feasibility distance tolerances. Each comparison is repeated for combinations of 20 random initial points, algorithms with basic setting and presence/absence of step back. The results are shown in Table 8.1.

Table 8.1 Scores of flexible against fixed maximum feasibility distance tolerance

Problem	Maximum fd	Sum of violated fd	Number of violations
adlittle	0.601	0.411	0.256
scfxm3	-0.446	-1	-0.774
bandm	0.988	-0.81	-0.649
bore3d	0.798	-0.881	0.536

brandy	1	-1	-0.458
bnl2	1	-1	-0.857
25fv47	0.476	-0.988	0.167
blend	0.357	0.155	0.393
CAPRI	0.75	-1	-0.667
crec	1	-1	0.548
CYCLE	0.036	-0.714	0.625
CZPROB	0.47	-1	-0.298
degen3	0.679	-1	0.321
FINNIS	0.821	0.774	-0.137
FIT1D	0.768	0	0
FIT1P	0.869	-0.244	0.738
GANGES	-0.107	-1	-0.845
GROW7	0.685	0	0
GROW15	-0.321	-0.321	-0.321
GROW22	-0.47	-0.5	-0.482
KB2	0.31	-0.571	-0.065
lotfi	0.988	-0.857	-0.589
MAROS	0.595	-0.976	0.018
modszk1	0.821	0.143	-1
nesm	0.339	0.083	-0.06
PILOT4	0.798	-1	0.268
PILOTJA	0.345	-0.786	-0.5
PILOTNOV	1	-1	-0.548
PILOTWE	0.512	-0.988	-0.857
RECIPELP	0.244	-0.845	-0.845
SC50A	0.464	0	0
SC105	0.702	0.012	0.012
SC205	0.702	0.702	-0.226
SCAGR7	0.893	0.726	0.446
SCAGR25	0.988	0.107	0.125
SCFXM1	0	-1	-0.179
SCFXM2	-0.065	-1	-0.411
SCTAP1	0.405	-0.881	-0.935
STOCFOR2	0.762	-0.738	-0.036
WOOD1P	0.125	-1	0.97
N better	*34*	*9*	*14*
N worse	*5*	*28*	*23*
all	*0.522*	*-0.525*	*-0.158*

8.1.2 Tangent Acceleration

This test compares use of tangent acceleration versus the algorithm without it. Each comparison is repeated for combinations of 20 random initial points, flexible/fixed feasibility distance tolerance and presence/absence of step back. The lowest result (number of iterations or maximum feasibility distance) among the multiple algorithms with tangent acceleration is compared with a single result of the algorithm without it. The results are shown in Table 8.2.

Table 8.2 Scores of presence against absence of tangent acceleration

Problem	Maximum fd	Sum of violated fd	Number of violations
adlittle	-0.143	0	0
scfxm3	-1	-1	-1
bandm	-0.524	0.714	0.143
bore3d	1	-0.81	-0.095
brandy	1	0.81	0.905
bnl2	0.952	0.905	0.714
25fv47	0.571	-1	-1
blend	0.476	0.429	0.143
CAPRI	0.81	-0.905	-0.905
crec	0.81	0.667	0.238
CYCLE	0.952	0.952	0.286
CZPROB	0.429	-0.524	-0.619
degen3	0.381	0.81	0.524
FINNIS	0.429	0.238	0.095
FIT1D	0.381	0	0
FIT1P	1	0.81	-0.81
GANGES	1	1	0.81
GROW7	0.952	0	0
GROW15	0.667	0.095	0.095
GROW22	0.857	0.667	0.667
KB2	-0.381	-0.952	-0.762
lotfi	1	1	-0.19
MAROS	1	-0.048	0.286
modszk1	1	1	1
nesm	1	0.81	0.81
PILOT4	1	0.143	0.19
PILOTJA	0.905	0.905	0.048
PILOTNOV	0.905	0.905	0.714
PILOTWE	1	0.905	0.714
RECIPELP	-0.905	-0.952	-0.952

SC50A	0.429	0	0
SC105	0.667	0	0
SC205	0.857	0.905	0.905
SCAGR7	0.619	0.619	0.571
SCAGR25	1	1	0.381
SCFXM1	-1	-1	-1
SCFXM2	-1	-1	-1
SCTAP1	-0.905	-0.905	-0.905
STOCFOR2	1	1	1
WOOD1P	1	-1	0.524
N better	*32*	*23*	*23*
N worse	*8*	*12*	*12*
all	*0.505*	*0.18*	*0.063*

8.1.3 Step Back

This test compares performance of the algorithms with step back task versus the algorithms without it. Each comparison is repeated for combinations of 20 random initial points, algorithms with basic setting and flexible/fixed feasibility distance tolerance. The results are shown in Table 8.3.

Table 8.3 Scores of presence against absence of the step back task

Problem	Maximum fd	Sum of violated fd	Number of violations
adlittle	0.905	0.69	0.69
scfxm3	1	0	1
bandm	0.952	0.762	1
bore3d	0.952	0.238	1
brandy	1	0.81	1
bnl2	0.429	0.905	1
25fv47	0	0	1
blend	0.571	0.571	0.952
CAPRI	0	0	1
crec	-0.857	-0.286	1
CYCLE	-0.905	-0.905	0.952
CZPROB	0	0	1
degen3	-0.143	-0.048	0.905
FINNIS	0.952	0.952	1
FIT1D	0.952	0	0
FIT1P	-0.143	0.524	1

GANGES	-0.19	0	1
GROW7	0.143	0.071	0.071
GROW15	0.429	0.429	0.429
GROW22	-0.048	-0.048	0
KB2	0.905	0.667	0.952
lotfi	-0.667	-0.667	1
MAROS	0.952	1	1
modszk1	-1	-1	0
nesm	0.952	0.667	0.762
PILOT4	0.286	0	1
PILOTJA	0.286	0.095	1
PILOTNOV	0.619	-0.048	1
PILOTWE	-0.667	-0.905	1
RECIPELP	0.952	0.81	0.952
SC50A	0.524	0	0
SC105	0	0	0
SC205	0	0	0.429
SCAGR7	0.762	0.762	1
SCAGR25	0.143	0.095	1
SCFXM1	0.905	0.19	1
SCFXM2	0.952	0.048	1
SCTAP1	0.143	0.048	0.905
STOCFOR2	-0.857	-0.381	1
WOOD1P	0	0.048	0.952
N better	*24*	*21*	*35*
N worse	*10*	*9*	*0*
all	*0.28*	*0.152*	*0.799*

8.1.4 Conclusions

- The results in Table 8.1 show that for most problems using flexible feasibility distance tolerance performs better than using fixed feasibility distance tolerance, though it does worsen the sum of feasibility distances and the number of violated constraints. All algorithms considered below use the flexible feasibility distance tolerance.
- The results in Table 8.2 show that for most problems using the tangent acceleration has higher performance than omitting it. However, the best among all algorithms with tangent acceleration is compared with only the single version without it. Also, step back has

another type of the movement vector acceleration as a possible alternative to the tangent acceleration. Therefore, both types of algorithms are considered further.

- The results in Table 8.3 show that for most problems using step back has higher performance than omitting it. Moreover, the algorithms without this procedure will not be efficient when faced with zigzagging. Therefore, step back is included in all of the algorithms considered below.

8.2 Determining the Best Minimum Set of New Algorithms

The best minimum sets among the 168 algorithms are evaluated using the optimization model described in Section 6.2. The optimization task is repeated for various values of *qp* corresponding to minimum proportion levels p_{min} of acceptable outputs for each problem. The values of p_{min} are based on $K = 20$ random initial points. The sizes of the best minimum sets of algorithms for various proportions p_{min} are shown in Table 8.4.

Table 8.4 Cardinalities of the best minimum sets of algorithms

Minimum fraction of acceptable outputs	1.0	0.9	0.8	0.7	< 0.6
Size of best minimum set	23	9	4	2	1

The list of settings of the new CC algorithm selected for the best minimum sets are shown in Table 8.5. The first four columns are algorithms and their settings (as listed in Table 7.1). The other four columns show the presence (1) or absence (0) of an algorithm in each set defined by the minimum proportion value p_{min} .

Table 8.5 Best minimum sets of algorithms

Algorithm	Selected violated constraints option	Algorithm settings	Movement direction option	*qp* = 0.9	*qp* = 0.8	*qp* = 0.7	*qp* < 0.6
No Tangent Acceleration	1	1	0	1	1	1	1
	1	1	1	1	1	0	0
	1	1	2	1	0	0	0
TUDA	0	2	0	1	0	1	0
	0	2	1	1	0	0	0
	0	2	3	1	0	0	0
	1	2	3	0	1	0	0
GDA	0	4	1	1	0	0	0
	0	4	3	1	0	0	0
	1	4	2	0	1	0	0
	2	4	0	1	0	0	0

Only the algorithms in the set corresponding to *qp* = 0.8 are used for further tests. The cardinalities of the sets for greater values of *qp* are significantly larger.

8.3 Selecting an Initial Point

This test shows the effect of selecting a specified initial point. It compares the best performance among the results obtained for 20 random initial points set within the variable bounds against a single result for the specified initial point set at the origin or as close to the origin as permitted by the bounds (thereafter referred to as the origin point). Each comparison is repeated for each algorithm from the best minimum set corresponding to *qp* = 0.8. The results are shown in Table 8.6.

Table 8.6 Scores of using a random initial point versus the fixed initial point at origin

Problem	Maximum fd	Sum of violated fd	Number of violations
adlittle	1	0.222	0.222
scfxm3	1	0.556	0.778
bandm	0.778	-0.778	0.556
bore3d	-0.778	0.556	1
brandy	1	0.111	0.111

bnl2	0.333	0	0.556
25fv47	1	1	0.556
blend	1	0	0
CAPRI	0.778	0.333	-0.778
crec	0.333	-0.778	1
CYCLE	-1	-1	-1
CZPROB	0.778	0.444	-0.111
degen3	-0.111	-0.556	-1
FINNIS	1	1	0.889
FIT1D	1	0	0
FIT1P	-0.778	-0.556	0.778
GANGES	0.333	0.333	-0.556
GROW7	1	0	0
GROW15	1	0	0
GROW22	1	0	0
KB2	-1	-1	-1
lotfi	1	1	0.778
MAROS	1	0.556	0.556
modszk1	1	1	1
nesm	1	0.778	0.778
PILOT4	-1	-1	1
PILOTJA	-0.333	0.111	0.556
PILOTNOV	0.333	-0.111	1
PILOTWE	0.556	-0.556	0.111
RECIPELP	-1	-1	-1
SC50A	1	0	0
SC105	1	0	0
SC205	-1	-1	-1
SCAGR7	0.889	0.778	0.778
SCAGR25	1	1	0.444
SCFXM1	0.778	0.778	0.111
SCFXM2	1	0.333	0.778
SCTAP1	-1	-1	-0.778
STOCFOR2	-1	-0.778	-0.111
WOOD1P	-1	-1	-0.556
N better	*28*	*18*	*22*
N worse	*12*	*14*	*11*
all	*0.347*	*-0.006*	*0.161*

The results show that the performance obtained for the randomly selected initial points is better than for the fixed initial point. However, the comparisons are performed for the best

among 20 random points against one fixed. The relatively good performance of the outnumbered origin point placement indicates that it should be included as a point during the final testing.

8.4 New Presolving Routine

The presolving method introduced in Section 5.2 is applied without repetition to the Netlib problems, after they have already been presolved by Cplex 12.6 [12], to see whether the new routine finds further model simplifications. This test is not used for tuning settings of the main algorithms, since it is applied prior to running the algorithms. Thus testing is applied to the full set of 114 Netlib models. Table 8.7 lists the problems which were further simplified by one iteration of the new presolver. A simplification means decreasing the number of variables and possibly decreasing the number of constraints.

Table 8.7 Netlib problems further simplified by the new presolver

	Problem	Number of variables	Removed variables	Number of constraints	Removed constraints
1	80bau3b	8249	161	1965	20
2	adlittle	94	16	66	0
3	afiro	14	1	14	0
4	AGG2	250	1	336	0
5	AGG3	249	1	337	0
6	bnl1	990	17	558	0
7	bnl2	2083	33	1167	10
8	boeing2	160	11	143	0
9	bore3d	60	1	74	4
10	CAPRI	203	1	162	3
11	D2Q06C	4554	5	2901	0
12	E226	245	10	167	0
13	FINNIS	362	46	311	67
14	GANGES	647	2	717	0
15	MAROS	809	1	645	0
16	pilot	3029	37	1498	0
17	PILOT87	4368	27	2084	72
18	SC105	30	1	32	0
19	SC205	60	1	65	0
20	SC50A	15	1	17	0

21	SCORPION	83	22	81	27
22	SCRS8	799	4	226	6
23	SCTAP1	339	12	269	18
24	SCTAP2	1326	47	977	62
25	SCTAP3	1767	62	1344	86

The results show that 25 of the 114 problems, previously presolved by Cplex, are further simplified by applying the new presolver. Iteratively applying the new presolver could produce further simplifications. While the simplifications produced by a single pass are relatively modest, this can significantly simplify the computation of the main algorithms, as described in Section 5.2. The complexity of a single new presolving task is $O(m \times n)$, e.g. equivalent to a single iteration of the main algorithms. The new presolver is thus adopted in all new algorithms. Repetitive application is recommended.

9. Experiments: Testing

9.1 Introduction to algorithm testing

After tuning the algorithms, the best minimum set is selected for testing, as shown in Table 9.1

Table 9.1 The selected best minimum set of new CC algorithms

Algorithm	Selected violated constraints option	Algorithm settings	Movement direction option
No Tangent Acceleration	1	1	0
	1	1	1
TUDA	1	2	3
GDA	1	4	2

All algorithms use the flexible feasibility point tolerance and step-back-and-expansion. The initial points are selected in both ways: randomly within the range 0 to 100 (or smaller to respect the variable bounds) and one prespecified point at the origin. Algorithms run after an initialization consisting of:

- The simple presolving tasks introduced in Section 7.3.1 and repeated for a maximum of 20 iterations.
- Transformation of arrays.
- Computing the tangent matrix and GD matrix.

The time required for the initialization steps is omitted in the timing statistics because the current prototype implementation of the tangent matrix calculation in R requires excessive computation time.

9.2 Comparison of New and Existing CC Algorithms

This test compares the simulated performance of the new and existing CC algorithms on four core concurrent computing. The comparison is performed for 20 random initial points (19 random and one fixed at origin), for which all new algorithms from the best minimum set (see Table 9.1), as well as all existing algorithms are run sequentially. For each initial point, the best performances among new and existing algorithms are compared. The comparison results are presented in Table 9.2.

Table 9.2 Comparison of the new and existing CC methods

Problem	Performance	sumfd	nviol	Time
afiro	1	0	0	0.15
bnl1	1	1	-0.8	0.1
agg	-1	0.3	0.05	-1
beaconfd	1	0.8	0.75	-0.9
degen2	1	0.1	-0.1	-0.9
boeing1	0.9	0.8	0.8	0.3
boeing2	0.8	-0.4	-0.8	-0.5
crea	1	0	0.2	-0.8
E226	0.9	0.9	-0.15	0.3
FF800	1	1	-1	1
GFRD	0.7	0.8	-0.1	0.9
ISRAEL	-0.1	-0.45	-0.65	-0.95
ken07	1	1	1	0.2
PEROLD	1	1	-0.8	1
QAP8	-0.45	0	0	-0.95
SC50B	0.85	0	0	0.05
STOCFOR1	1	-0.4	-0.75	0
SCORPION	1	1	1	-1
SCRS8	-0.3	0.4	-1	0.4
SCSD1	1	0.15	0.15	0.2
SCSD6	0.5	0.1	-0.65	0.1
SCSD8	0.5	0.1	-0.55	0
SCTAP2	1	-0.35	-0.95	-0.8
SCTAP3	1	-0.75	-0.95	-0.9
SEBA	1	0.7	-0.3	0.9
SHARE1B	1	1	-1	-0.3
SHARE2B	0.6	0.5	0.6	-0.5
SHELL	1	1	0.1	-1

SHIP04L	-0.4	-0.7	-0.8	0
SHIP04S	-0.3	-0.55	-0.85	0.1
SHIP08L	0.1	-0.7	-1	-0.1
SHIP08S	0.1	0.15	0.05	-0.5
SHIP12L	0.8	-0.9	-0.9	-0.4
SHIP12S	0.5	0.05	-0.95	-0.9
SIERRA	1	1	1	-0.3
STAIR	1	1	0.2	0.1
STANDATA	0.3	0.8	-0.6	-0.3
STANDGUB	0.3	0.8	-0.6	-0.8
STANDMPS	0.8	0.7	-1	-0.2
TUFF	1	1	-1	-1
VTP-BASE	1	1	0.55	-0.6
WOODW	1	0.5	-0.9	0.2
80bau3b	1	1	1	1
D2Q06C	1	0	-0.1	0.4
D6CUBE	0	-0.1	-0.7	-0.6
pilot	1	0.4	-0.5	1
N better	*39*	*31*	*14*	*19*
N worse	*6*	*10*	*29*	*24*
all	*0.654*	*0.342*	*-0.283*	*-0.17*

The “performance” column shows the average score in terms of the difference in the numbers of iterations if successful, or difference in maximum feasibility distances if unsuccessful (+1 if the new algorithms are better, -1 if the old algorithms are better). A positive average score indicates net better performance for the new algorithms over the set of tests, while a negative average score indicates net better performance for the existing algorithms. For most of the Netlib problems, the new CC algorithms have better performance than the existing CC algorithms. As was predicted, the existing algorithms are faster, but the time differences are not significant, as expected. This is because the common part in all of the algorithms (new and existing) is the most time-consuming: calculating the feasibility distances over all of the constraints. The new CC algorithms have more complex routines dealing with the violated constraints, but they are a fraction of the total. Also, different algorithms produce different numbers of violated constraints,

which impacts computation time. If we count only significant relative time differences, which are outside of the interval [80%, 120%], only one problem has non-zero average time difference.

Table 9.3 shows the problems that were successfully solved by the new and existing CC algorithms. The best 4 new algorithms and 4 existing algorithms were computed at 5 different initial points each. Large problems having more than 1000 variables or more than 1000 constraints are shown in boldface.

Table 9.3 The successfully solved problems

Problem	New Algorithms	Existing Algorithms
afiro	1	1
ken07	1	0
QAP8	1	1
SC50B	1	1
SCORPION	1	0
SCSD1	1	1
SHIP04S	1	1
SHIP08S	1	1
SHIP12S	1	1
80bau3b	1	0

Some conclusions follow:

- The new CC algorithms have more successes than the existing algorithms (10 vs. 7).
- The size of a problem is not a significant factor in success.

9.3 Expected Performance of Concurrent Computing of New CC Algorithms

The expected performance of concurrent computing of multiple algorithms is simulated by sequential computing of these algorithms. When the number of cores is large enough to concurrently compute all runs, i.e. some combination of the algorithms and various initial points, the best run is the expected performance. Where there are fewer cores than runs, the cores cannot

cover all runs, and the expected performance is a random variable depending on the probability of selecting each run for inclusion in the set of runs.

9.3.1 Number of Cores is Equal to Number of Algorithms

This test provides the simulation of concurrent computing of the best minimum set of four new CC algorithms shown in Table 9.1. The performance is calculated for four-core concurrent computing. Each algorithm from the set is computed sequentially one time starting from an initial point generated randomly or fixed at origin. The best performance is selected. This process is repeated 30 times in order to produce performance statistics; these are summarized in Table 9.4. Each cell shows the mean and standard deviation over the 30 trials. *Niter* is the number of

Table 9.4 Means and standard deviations of expected performance of 4 algorithms on 4 cores

Problem	niter	maxfd	sumfd	nviol	Time (s)
afiro	78.1, 23.2	1e-05, 0	1e-05, 0	0, 0	0.00107, 0.00406
bnl1	5000, 0	0.808, 0.139	123, 41.7	913, 55.5	0.714, 0.233
agg	5000, 0	523, 42.9	4840, 447	65.2, 3.79	0.151, 0.00761
beaconfd	5000, 0	2.37e-05, 2.42e-06	0.000279, 7.73e-05	17.8, 4.73	0.197, 0.0203
degen2	5000, 0	0.00302, 0.000763	0.513, 0.0956	442, 16.3	0.485, 0.124
boeing1	5000, 0	0.571, 0.0684	10.5, 4.94	137, 38	0.235, 0.0388
boeing2	5000, 0	0.459, 0.1	4.91, 1.09	40.2, 6.36	0.0808, 0.0109
crea	5000, 0	1.07, 0.207	1320, 558	3210, 270	1.98, 0.444
E226	5000, 0	0.102, 0.0549	3.52, 2.4	168, 22.8	0.224, 0.0314
FF800	5000, 0	178, 4.35	19000, 4860	808, 31.2	0.75, 0.0432
GFRD	5000, 0	16.3, 1.32	112, 70.9	638, 103	0.337, 0.0155
ISRAEL	3880, 1620	0.0914, 0.116	0.61, 1.08	12.9, 16.3	0.0778, 0.0331
ken07	3950, 781	1.13e-05, 5.67e-06	0.000226, 0.00117	12.8, 68.2	1.04, 0.293
PEROLD	5000, 0	2.36, 0.15	579, 33.5	993, 31.9	0.638, 0.0722
QAP8	584, 127	1e-05, 0	1e-05, 0	0, 0	0.156, 0.0386
SC50B	339, 881	9.67e-06, 1.83e-06	9.67e-06, 1.83e-06	0, 0	0.00103, 0.00393
STOCFOR1	5000, 0	0.0244, 0.0109	0.412, 0.237	70.9, 6.92	0.0605, 0.0162

SCORPION	4530, 617	1.44e-05, 7.88e-06	0.000148, 0.000241	8.73, 13.9	0.153, 0.0291
SCRS8	5000, 0	0.548, 0.141	129, 72.2	1030, 33.7	0.492, 0.113
SCSD1	2130, 457	1e-05, 0	1e-05, 0	0, 0	0.165, 0.0383
SCSD6	5000, 0	0.00822, 0.00813	0.496, 0.474	148, 5.18	0.55, 0.0386
SCSD8	5000, 0	0.0253, 0.00876	4.41, 1.74	434, 174	1.22, 0.138
SCTAP2	4840, 901	0.16, 0.0387	59.7, 15.8	1080, 205	0.665, 0.167
SCTAP3	4840, 903	0.164, 0.0396	82.4, 21.9	1460, 278	0.857, 0.225
SEBA	5000, 0	6.51, 0.207	531, 201	521, 14.7	0.644, 0.16
SHARE1B	5000, 0	47.1, 1.05	2510, 119	197, 3.69	0.233, 0.0376
SHARE2B	5000, 0	0.0561, 0.0209	0.708, 0.33	46.6, 6.11	0.0684, 0.0145
SHELL	5000, 0	0.846, 1.15	294, 404	786, 211	0.777, 0.208
SHIP04L	5000, 0	0.00161, 0.00102	0.346, 0.206	541, 172	0.728, 0.114
SHIP04S	5000, 18	0.000324, 0.000328	0.0413, 0.0509	241, 103	0.507, 0.0776
SHIP08L	5000, 0	0.0034, 0.00235	1.51, 1.26	1740, 375	1.39, 0.225
SHIP08S	4740, 424	6.37e-05, 6.75e-05	0.0114, 0.0161	202, 224	0.585, 0.0819
SHIP12L	5000, 0	0.0206, 0.00902	12.3, 5.57	2010, 473	1.73, 0.238
SHIP12S	5000, 0	0.00713, 0.00535	1.07, 1.22	449, 100	0.779, 0.101
SIERRA	5000, 0	2.59e-05, 5.02e-06	0.000605, 0.000487	37.2, 29.2	0.782, 0.0619
STAIR	5000, 0	0.0544, 0.0362	4.61, 4.49	323, 25.8	0.359, 0.0577
STANDATA	5000, 0	0.766, 0.553	166, 147	956, 104	0.476, 0.128
STANDGUB	5000, 0	0.818, 0.59	179, 159	941, 110	0.544, 0.145
STANDMPS	5000, 0	0.786, 0.563	211, 162	979, 87	0.649, 0.199
TUFF	5000, 0	0.548, 0.0476	93.1, 6.29	599, 8.28	0.609, 0.158
VTP-BASE	5000, 0	14.3, 1.05	799, 142	112, 11	0.0802, 0.0222
WOODW	5000, 0	0.305, 0.118	451, 87.8	6530, 193	5.12, 0.607
80bau3b	525, 179	1e-05, 0	1e-05, 0	0, 0	0.133, 0.0447
D2Q06C	5000, 0	28.4, 1.05	11200, 5180	3350, 176	4.11, 0.1
D6CUBE	5000, 0	0.0971, 0.0378	213, 129	4780, 583	5.6, 1.13
pilot	5000, 0	1.65, 0.122	1380, 82.8	2280, 88.3	2.25, 0.133

Among 46 problems, 11 problems have at least one success in 30 runs before reaching the maximum number of iterations. The average number of iterations for these problems is lower than 5000. Other 8 problems have relatively low maximum feasibility distance, and probably could be successfully finished if the maximum number of iterations were slightly larger. Some of

the 11 problems have more than 1000 constraints or bounds. It can be concluded that the internal structure of a problem, as determined by the set of constraints, can have a greater impact on success in finding a feasible point than the dimensions of a problem.

Table 9.5 shows how the main metrics of infeasibility were reduced after running the 4 new CC algorithms concurrently. The values of the metrics at the initial points are shown in columns 2-4 while the ratio of the initial to the final values is shown in columns 5-7 ("feasible" is shown if a feasible point is reached, since the ratio is infinity).

Table 9.5 Ratios of infeasibility metrics before and after computing

Problem	maxfdInit	sumfdInit	nviolInit	maxfdRatio	sumfdRatio	nviolRatio
afiro	135.3	619.2	13	Feasible	Feasible	Feasible
bnl1	378.8	33300	410	473	283.4	0.563
agg	616300	3530000	54	1241	854.6	1.136
beaconfd	849.6	8878	81	3.60E+07	43460000	6.543
degen2	426.7	17850	302	161500	47980	0.795
boeing1	945.7	6976	69	1565	1042	0.99
boeing2	955.7	4619	33	1761	850.2	1.352
crea	759.6	11730	344	783.9	25.8	0.125
E226	312.8	6637	94	3645	2852	0.665
FF800	5089	57080	391	29.06	3.415	0.491
GFRD	26890	133800	562	1636	1797	0.89
ISRAEL	357	4540	52	1477	4597	4.215
ken07	200.3	67040	1417	Feasible	Feasible	Feasible
PEROLD	12830	48320	568	5301	85.1	0.6
QAP8	190.5	92060	867	Feasible	Feasible	Feasible
SC50B	74.62	887.1	31	Feasible	Feasible	Feasible
STOCFOR1	139.7	2529	65	4595	8488	1.08
SCORPION	137	10730	257	Feasible	Feasible	Feasible
SCRS8	330.6	25680	363	609.5	201.5	0.371
SCSD1	70.9	1671	73	Feasible	Feasible	Feasible
SCSD6	76.8	3219	140	12590	11210	0.953
SCSD8	83.92	8850	377	2977	1920	0.944
SCTAP2	152.4	39550	515	1187	2553	0.649
SCTAP3	154.3	53000	706	988.1	2425	0.748
SEBA	179.6	37960	447	27.4	98.96	0.865
SHARE1B	1673	18360	92	35.08	7.563	0.574
SHARE2B	153.9	3374	43	2648	4870	1.16

SHELL	25530	401400	714	8546000	1711000	1.106
SHIP04L	398	39680	280	358400	225500	0.844
SHIP04S	314.2	26750	205	Feasible	Feasible	Feasible
SHIP08L	416.2	73050	509	191200	119600	0.601
SHIP08S	273.8	31050	259	Feasible	Feasible	Feasible
SHIP12L	374	92380	632	28590	17090	0.423
SHIP12S	223.7	35630	317	Feasible	Feasible	Feasible
SIERRA	1459	35600	740	47890000	86080000	27.95
STAIR	190.1	15440	309	6171	7285	1.005
STANDATA	1146	17060	235	2281	153.3	0.32
STANDGUB	1146	17060	235	2281	153.3	0.32
STANDMPS	1146	29540	343	2686	279.1	0.415
TUFF	533.9	22720	260	950.9	242.3	0.458
VTP-BASE	2149	18090	72	145.9	21.68	0.649
WOODW	1086	84650	944	6507	399.5	0.187
80bau3b	885.3	23220	443	Feasible	Feasible	Feasible
D2Q06C	3478	125400	1759	119.7	45.29	0.591
D6CUBE	2590	35380	380	60290	791.4	0.114
pilot	638.7	99740	853	386.4	75.13	0.384

All problems show significant reduction of the maximum feasibility distance and sum of feasibility distances, but the number of violated constraints increases for most of them. This is because movements reducing the maximum feasibility distance often increase the number of relatively small violations. In addition, this is the number of violations associated with the lowest maximum feasibility distance result in the run, which is not likely to be the smallest number of constraint violations seen in the run.

9.3.2 Fewer Processors than Runs

This test simulates concurrent operations of the best minimum set of four new CC algorithms shown in Table 9.2. Each algorithm from the set is computed sequentially five times, hence there are 20 runs starting from 20 different initial points generated randomly or fixed at origin. If concurrent computing is performed on fewer than 20 cores, then not all 20 runs can be executed

simultaneously. We expect that each run may or may not be selected with the same probability. The analytical calculation of the concurrent performance was described in Section 6.4.

Figures 9.1 and 9.2 show the dependency of the performance of different problems on the number of processors using for concurrent computing. Figure 9.1 shows the plots of the number of iterations for the 12 problems on which the set of new CC algorithms were successful (terminated with all constraint violations within the feasibility distance tolerance). Figure 9.2 shows the plots of the lowest maximum feasibility distances for the 34 problems for which the set of new CC algorithms failed.

The figures show that an increase in number of processors improves the performance of the concurrent implementation. The highest impact of the increase is obtained for the first few added processors beyond 1. The extreme example is the SEBA problem. Among 20 runs one of them completely failed (the iteration process was stopped due to very large feasibility distances). The expected maximum feasibility distance is also very high because with probability $1/20$ the computation will fail. Adding another core will override this problem, because any other run will finish earlier.

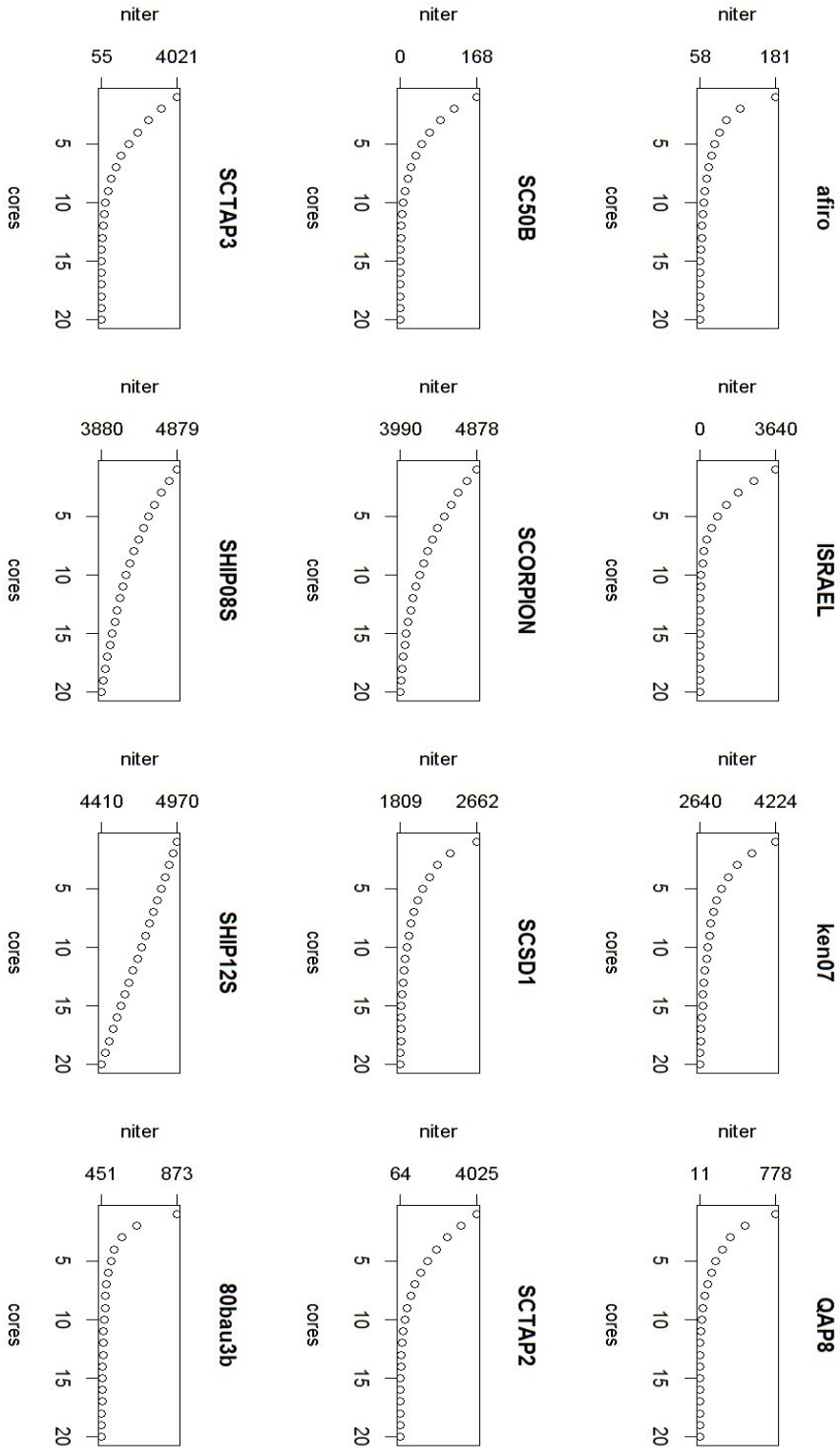
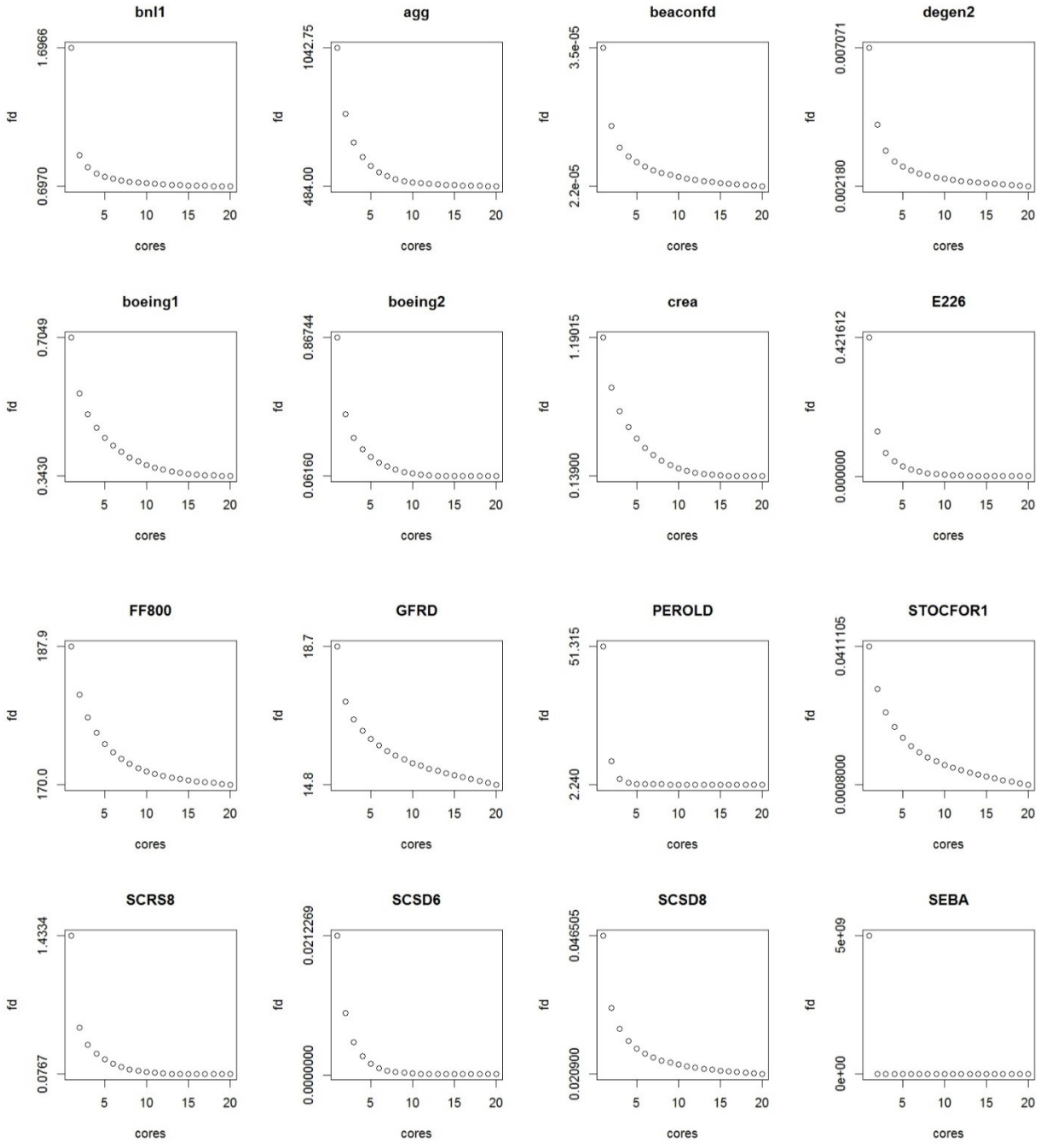
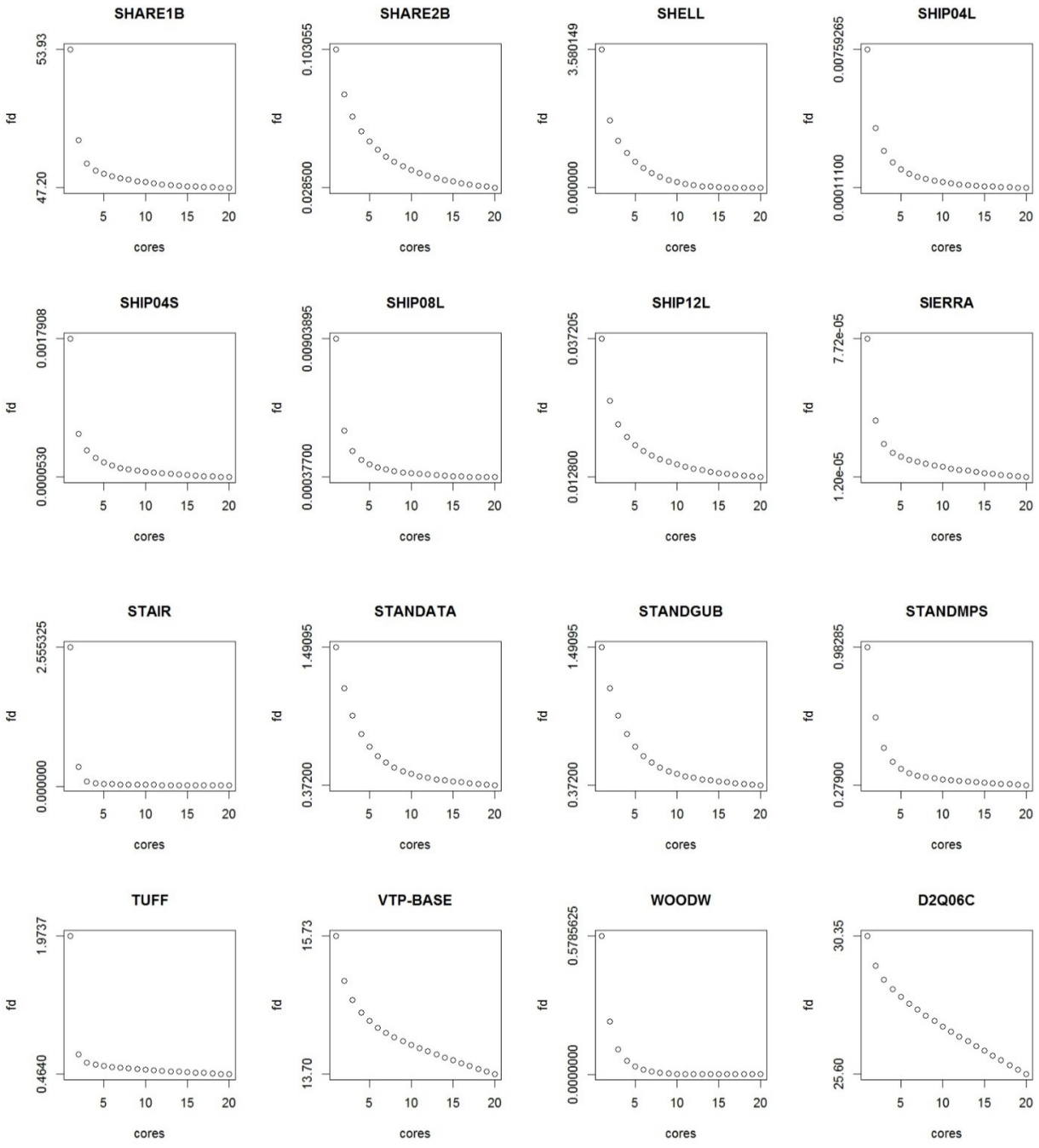


Figure 9.1 Number of iterations needed to find a feasible point by concurrent computing using the best minimum set of 4 new CC algorithm


bnl1
1.6966
0.6970
agg
1042.75
484.00
beaconfd
3.5e-05
2.2e-05
degen2
0.007071
0.002180
boeing1
0.7049
0.3430
boeing2
0.86744
0.06160
crea
1.19015
0.13900
E226
0.421612
0.000000
FF800
187.9
170.0
GFRD
18.7
14.8
PEROLD
51.315
2.240
STOCFOR1
0.0411105
0.0008000
SCRS8
1.4334
0.0767
SCSD6
0.0212269
0.0000000
SCSD8
0.046505
0.020900
SEBA
5e+09
0e+00
fd
cores
5
10
15
20


SHARE1B
fd
53.93
47.20
5 10 15 20
cores
SHARE2B
fd
0.103055
0.028500
5 10 15 20
cores
SHELL
fd
3.580149
0.000000
5 10 15 20
cores
SHIP04L
fd
0.00759265
0.00011100
5 10 15 20
cores
SHIP04S
fd
0.0017908
0.0000530
5 10 15 20
cores
SHIP08L
fd
0.00903895
0.00037700
5 10 15 20
cores
SHIP12L
fd
0.037205
0.012800
5 10 15 20
cores
SIERRA
fd
7.72e-05
1.20e-05
5 10 15 20
cores
STAIR
fd
2.555325
0.000000
5 10 15 20
cores
STANDATA
fd
1.49095
0.37200
5 10 15 20
cores
STANDGUB
fd
1.49095
0.37200
5 10 15 20
cores
STANDMPS
fd
0.98285
0.27900
5 10 15 20
cores
TUFF
fd
1.9737
0.4640
5 10 15 20
cores
VTP-BASE
fd
15.73
13.70
5 10 15 20
cores
WOODW
fd
0.5785625
0.0000000
5 10 15 20
cores
D2Q06C
fd
30.35
25.60
5 10 15 20
cores

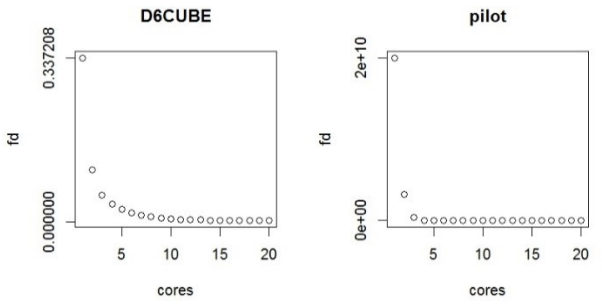


Figure 9.2 Lowest values of feasibility distances after 5000 iterations of concurrent computing using the best minimum set of 4 new CC algorithms

9.4 Scaling

The new CC algorithms were applied to the Netlib problems, some of which have thousands of variables and constraints. Some linear systems can have much larger dimensions, such as millions of variables and constraints. An important question is whether the new algorithms will scale sufficiently to handle very large models.

The overall complexity of the new algorithms is O($\mu \times m \times n$). However the current prototype implementation is lacking, especially in the initialization step, which is slow even for low-dimensional systems, but can be significantly simplified. The inefficient implementation of the calculation of the tangent matrix is the main opportunity for improvement. Instead of looping over each pair of rows, which has a very large complexity, a column-based structure can be used. Note that for sparse structures, the total number of non-zero elements has a larger impact on computing time than the number of variables, and most models will be very sparse. In addition, the algorithm should be implemented in a compiled language, like C, rather than the current implementation in the interpreted language R.

There are no theoretical barriers to scaling the algorithms to handle very large problems. Adding computing cores will also help scale the algorithm by decreasing the computation time by allowing more different algorithms to run from more initial points. This increases the chances of an early successful termination.

9.5 Conclusions

The tests performed in this section lead to the following conclusions:

- The new CC algorithms generally produce smaller feasibility distances in fewer iterations. However, all new CC algorithms, compared to the existing CC algorithms, need more time for a single iteration, which has the same complexity with larger constants. As a result, the existing CC algorithms are faster, though the time differences are not significant. In other words, the test shows that the new CC algorithms have better average performance obtained in slightly more time. Using maximum computing time instead of maximum number of iterations would provide a fairer comparison, however it is much more difficult to estimate a reasonable time limit in advance.
- More accurate comparison of computing times should include the initialization step since computing the tangent matrix is time-consuming, and this step is not needed for the existing CC algorithms. This comparison can be performed only after significant improvement of the implementation for computing the tangent matrix.
- The new algorithms successfully solve almost 25% of the testing problems within 5000 iterations. Another 15% of the problems fail with relatively low values of the maximum feasibility distance. The remaining problems may not be successfully finished without significant increase of the number of iterations. The existing algorithms successfully solve almost 15% of the testing problems. The difference in percentages of success depends on the maximum number of iterations.
- The new algorithms generally provide massive improvements in the maximum feasibility distance and the sum of the feasibility distances, but often increase the number of violated constraints.

- Simulation of the expected performance of a concurrent implementation shows that increasing the number of cores is initially very effective. This filters out the extremely inefficient runs, if they do not occur frequently.

10. Conclusions and Further Work

10.1 Conclusions

The main conclusions of this research are the following:

- Overall, the new CC algorithms are better than the existing CC algorithms.
- The new feasibility-seeking presolver simplifies the problems in some cases. Considering its fast execution, its application is recommended.
- Various types of tangent-based acceleration are recommended (except the MAUD algorithm).
- Algorithms using relaxation and flexible distance tolerance have better performance.
- Mixing initial points generated randomly and at the origin improves the performance of the algorithms.
- A concurrent implementation of the new algorithms improves overall performance.
- An effective concurrent implementation does not need a large number of cores

10.2 Contributions

The main contribution of this research is the development of improved CC algorithms that generally outperform the existing algorithms in finding approximate feasible points for systems of linear constraints. The new algorithms are better due to multiple novel improvements:

- Presolving is improved by adding a new technique targeted to finding a feasible point.
- Tangent-based acceleration increases the lengths of feasibility vectors in a new way, depending on their ill-conditioning, as measured by the angle between them. As a result, the feasibility vectors belonging to more ill-conditioned constraints see a larger increase in their length.

- The step back followed by expansion of the movement vector consists of two tasks. The step back improves the direction of the next movement vector so that it points more directly towards a feasible region. The expansion increases the length of the next movement in this improved direction.
- Selection of subsets of the violated constraints for improvement.
- Improvement of the movement direction obtained by modifying the calculation of feasibility vector.
- An algorithm for estimating computing time on a small number of processors

10.3 Further Work

The new CC algorithms can be improved in multiple ways, as outlined below.

The new presolving technique can be iteratively applied together with multiple existing LP presolving techniques to reduce the problem dimensionality and thereby simplify finding a feasible point.

The expansion routine has many possible improvements. It uses empirical values that can be adjusted in multiple ways. Starting from Step 6, Algorithm 5.10 can be modified to increase expansion, so that any increases in the feasibility distances for the worsened constraints would not be significant.

The implementation of the algorithms can be improved in order to make them faster. An example of an improvement is that a pair of constraints created from a single equality constraint can be checked for violation in a more efficient way: at any iteration, both constraints cannot be violated simultaneously.

It is possible to improve the performance of the concurrent implementation by increasing the variability among its runs. This may improve the runs which are relatively better, and, hence it will finish faster. On the other hand, worsening other results will not impact the performance of a concurrent implementation, if the number of cores is large enough. Another possibility is changing the strategy for generating initial points, which can increase the variability of performances of different runs.

The performance can also be improved by applying the best minimum set containing both new and existing CC algorithms. Their different approaches may increase variability among their concurrent run results, giving better overall performance.

Concurrent computing of the algorithms can be improved. The current implementation of an expansion task can significantly reduce the maximum feasibility distance, as well as significantly increase it. If it was increased and was not improved during a given number of iterations, the point can be returned to its saved best position. The solution process can continue from there by applying another algorithm, or by adjusting the settings of the current algorithm.

The new CC algorithm software can be applied to very large problems after the initialization is significantly improved (see Section 9.4). Larger problem sizes will increase the computation time of a single iteration, but there may be no need to increase the maximum number of iterations.

The possibility of applying the new CC methods for nonlinear systems should be investigated. The possible restrictions are as follows:

- The tangent acceleration will need more computation time. For nonlinear systems, the tangent between nonlinear constraints depends on the current position of the point

because the linear approximation is based on the gradient. As a result, the tangent values will have to be computed at each iteration, which will significantly increase the computation time. This is feasible when there is only a small number of nonlinear constraints.

- The step-back routine can be applied, but the expansion task should be redesigned, because it is based on ratios applicable only for linear constraints.
- The tangent acceleration will need more computation time. For nonlinear systems, the tangent between nonlinear constraints depends on the current position of the point because the linear approximation is based on the gradient. As a result, the tangent values will have to be computed at each iteration, which will significantly increase the computation time. This is feasible when there is only a small number of nonlinear constraints.
- The step-back routine can be applied, but the expansion task should be redesigned, because it is based on ratios applicable only for linear constraints.

References

[1] A.Basu, J.A.De Loera and M.Junod. On Chubanov's method for Linear Programming

Optimization and Control Math. Programming 134(3):533–570, 2013.

[2] E.D.Andersen and K.D.Andersen. Presolving in linear programming. *Mathematical Programming* 71 (1995) 221-245.

[3] J-D.Boissonnat and M.Yvnec. Algorithmic Geometry*, Cambridge University Press,* 1998.

[4] Y.Censor. Row-Action Methods for Huge and Sparse Systems and Their Applications. *SIAM Review*, Vol. 23, No. 4 (Oct., 1981), pp. 444-466

[5] Y.Censor, A.Cegielski. Projection methods: an Annotated Bibliography of Books and Revievs. *Optimization* , Volume 64, Issue 11, 11/2015.

[6] Y.Censor, W.Chen, P.L Combettes, R.Davidi and G.T.Herman, On the effectiveness of projection methods for convex feasibility problems with linear inequality constraints, *Computational Optimization and Applications* 51 (2012), 1065—1088.

[7] Y.Censor, D.Gordon, R.Gordon. 2001. Component averaging: An efficient iterative parallel algorithm for large and sparse unstructured problems. *Parallel Comput.* 27 777–808.

[8] Y.Censor and S.A.Zenios, Parallel Optimization: Theory, Algorithms, and Applications, *Oxford University Press*, New York, NY, USA, 1997.

[9] Y.Censor and Y. Zur. Linear Superiorization for Infeasible Linear Programming. *Lecture Notes in Computer Science* (LNCS), Vol. 9869, 2016, Springer International Publishing, pp. 15-24.

[10] J.W.Chinneck. The constraint consensus method for finding approximately feasible points in nonlinear programs. *INFORMS Journal on Computing,* 16(3) :255- 265, Summer 2004.

[11] S.Chubanov. A strongly polynomial algorithm for linear systems having a binary solution. *Mathematical Programming*, 134:533{570, 2012.

[12] Cplex optimizer. Source:

[13] G.Cimmino, Calcolo approssimato per soluzioni dei sistemi di equazioni lineari, *La Ricerca Scientifica* XVI, Series II, Anno IX1 (1938), 326—333 (in Italian).

[14] Code::Blocks: a *free C, C++ and Fortran IDE. Source: http://www.codeblocks.org/.*

[15] N.I.M.Gould. How good are projection methods for convex feasibility problems? *Comput. Optim. Appl*. 40, 1–12 (2008).

[16] A.Grama, A.Gupta, G.Karypis, V.Kumar. Introduction to Parallel Computing. *Addison-Wesley*, Harlow, England, 1994

[17] W.Ibrahim and J.W.Chinneck. Improving solver success in reaching feasibility for sets of nonlinear constraints. *Computers and Operations Research,* 2005.

[18] S.Kaczmarz, Approximate solution of systems of linear equations, *International Journal of Control* 57 (1993), 1269—1271 Translated from the German original of 1933.

[19] N.Karmarkar. A New Polynomial Time Algorithm for Linear Programming. *Combinatorica*, 1984, Vol **4**, nr. 4, p. 373–395.

[20] M.D.McKay, W.J.Conover, and R.J.Beckman. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics,* 21(2):239-245, May 1979.

[21] B.A.Murtagh and M.A.Saunders. MINOS 5.5 user's guide. Technical Report SOL 83-20R, *Systems Optimization Laboratory*, Stanford University, July 1998.

[22] Netlib LP problems. Source: *http://www.netlib.org/lp/data/.*

[23] J. von Neumann. Functional Operators, Volume II: The Geometry of Orthogonal Spaces, Annals of Mathematics Studies, Volume AM-22, 1950, Princeton University Press, Princeton, NJ, USA. Reprint of mimeographed lecture notes first distributed in 1933.

[24] R.L.Rardin. Optimization in Operations Research. Prentice Hall, New Jersey, USA, 1998.

[25] The R foundation, R: The R project for statistical computing (online). Source: *https://www.r-project.org/.*

[26] W.L.Winston. Operational Research: Applications and Algorithms. PWS Publishers, 1987.

[27] Y.Xiao, D.Michalski, J.Galvin, and Y. Censor. The least-intensity feasible solution for aperture-based inverse planning in radiation therapy. *Annals of Operations Research*, 119(1-4): 183-203, March 2003.