A class of trust region methods for nonlinear network optimization problems, including some numerical experiments (revised)

by A. Sartenaer*

July 1, 1993

Abstract. We describe the results of a series of tests upon a class of new methods of trust region type for solving the nonlinear network optimization problem. These methods are characterized by the use of the infinity norm and of inexact projections on the network constraints. The results are encouraging and show that the methods are particularly useful in solving large-scale nonlinear network optimization problems.

* **Department of Mathematics, Facultés Universitaires ND de la Paix, Namur, Belgium**

Keywords : Nonlinear optimization, nonlinear network optimization, trust region methods, truncated Newton methods, numerical results.

A CLASS OF TRUST REGION METHODS FOR NONLINEAR NETWORK OPTIMIZATION PROBLEMS, INCLUDING SOME NUMERICAL EXPERIMENTS (REVISED)

by A. Sartenaer*

Report 93/21 July 1, 1993

Abstract. We describe the results of a series of tests upon a class of new methods of trust region type for solving the nonlinear network optimization problem. These methods are characterized by the use of the infinity norm and of inexact projections on the network constraints. The results are encouraging and show that the methods are particularly useful in solving large-scale nonlinear network optimization problems.

* **Department of Mathematics, Facultés Universitaires ND de la Paix, Namur, Belgium**

Keywords : Nonlinear optimization, nonlinear network optimization, trust region methods, truncated Newton methods, numerical results.

1 Introduction

We consider the problem:

$$\begin{array}{ll} \min_{x \in \mathbf{R}^n} & f(x) \\ \text{subject to} & Ax = b \\ & l \leq x \leq u, \end{array} \tag{1.1}$$

where $f : \mathbf{R}^n \to \mathbf{R}$ is a twice continuously differentiable partially separable function, A is a $m \times n$ node-arc incidence matrix, $b \in \mathbf{R}^n$ and satisfies $\sum_{i=1}^m b_i = 0$, and l and $u \in \mathbf{R}^n$.

Many algorithms for solving the nonlinear network problem (1.1) have been proposed (see [1], [3], [10], [11], [13], [21] and [22] for instance), most of them being of the active set variety. In particular, a sequence of problems are solved for which a subset of the variables (the active set) are fixed at bounds and the objective function is minimized with respect to the remaining variables. Such algorithms typically use linesearches to enforce convergence. A significant drawback of these methods, especially for large-scale problems, is that the active sets are allowed to change slowly and many iterations are necessary to correct a bad initial choice.

In this paper, we propose a new algorithm of *trust region* type that allows rapid changes in the active set. This algorithm is an adaptation of the one proposed by Conn, Gould, Sartenaer and Toint in [6] for which we have already produced a general convergence theory. At each iteration of the algorithm, we define a local model of the objective function at the current iterate, x_k say, and a *region* surrounding x_k where we *trust* this model. The algorithm then finds, in this region, a candidate for the next iterate that sufficiently reduces the value of the model. If the function value calculated at this point matches its predicted value closely enough, then the new point is accepted as the next iterate and the trust region is possibly enlarged. Otherwise, the point is rejected and the trust region size is decreased.

The determination of a candidate for the next iterate requires the computation of a *Generalized Cauchy Point* which expands the notion of a Cauchy Point to problems with general convex constraints (see [6]). This has the double advantage of allowing significant changes in the active set at each iteration and permitting the extension of well-known convergence results for trust region methods applied to unconstrained problems (see [18]) and to simple bound constrained problems (see [4]).

Furthermore we exploit the structure of network constraints in (1.1) by combining a data structure of the type proposed by Bradley, Brown and Graves [2] with a partition of the variables similar to that proposed by Murtagh and Saunders [19] and implemented in MINOS.

Section 2 of the paper gives a general introduction to the framework of our algorithm, together with a detailed description of the computation of a Generalized Cauchy Point and of a candidate for the next iterate. The optimality conditions and the specific algorithm are also presented in this section. Section 3 reports and comments on some numerical experiments, and includes a comparison with an existing available specialized software for the same problem. Finally some conclusions and perspectives are outlined in Section 4.

2 Description of the algorithm

2.1 The basic algorithm

As already mentioned, our algorithm is of trust region type and the description given here is a special case of the general framework presented in [6], adapted to the solution of problem (1.1). We first introduce the following concepts. The *feasible region* for problem (1.1) is the polyhedral set

$$X = \{x \in \mathbf{R}^n | Ax = b \text{ and } l \leq x \leq u\},$$

and any point x in the feasible region is called *feasible*. We define the *active set with respect to the vectors l and u at the feasible point x* as the index set

$$\mathcal{A}(x, l, u) = \{i \in \{1, \ldots, n\} | \, [x]_i = [l]_i \text{ or } [x]_i = [u]_i\},$$

where $[v]_i$ denotes the ith component of the vector v.

At the kth stage of the algorithm, we suppose that we have a feasible point x_k, the exact gradient $\nabla f(x_k)$ (denoted g_k) and the exact Hessian $\nabla^2 f(x_k)$ (denoted H_k) of the objective function at x_k. We also require a scalar $\Delta_k > 0$ for the trust region radius, and choose the quadratic model of the form

$$m_k(x_k + s) \stackrel{\text{def}}{=} f(x_k) + g_k^T s + \frac{1}{2} s^T H_k s$$

to approximate the objective function around x_k. A trial feasible step s_k is then computed by approximately solving the trust region problem

$$\begin{array}{ll} \min_{s \in \mathbf{R}^n} & m_k(x_k + s) \\ \text{subject to} & As = 0 \\ & l \leq x_k + s \leq u \\ \text{and} & \|s\| \leq \Delta_k, \end{array} \tag{2.1}$$

where $\|.\|$ is a suitable chosen norm. The updates of the iterate x_k and of Δ_k are done using the same criteria of acceptance as in trust region methods for unconstrained or bound constrained minimization (see [18] and [4]). That is,

$$x_{k+1} = \begin{cases} x_k + s_k & \text{if } \rho_k > \eta_1 \\ x_k & \text{if } \rho_k \leq \eta_1 \end{cases}$$

and

$$\Delta_{k+1} = \begin{cases} 2\Delta_k & \text{if } \rho_k \geq \eta_2 \\ \Delta_k & \text{if } \eta_1 < \rho_k < \eta_2 \\ \frac{1}{2}\min(\|s_k\|, \Delta_k) & \text{if } \rho_k \leq \eta_1, \end{cases} \tag{2.2}$$

where

$$\rho_k = \frac{f(x_k) - f(x_k + s_k)}{m_k(x_k) - m_k(x_k + s_k)}$$

represents the ratio of the achieved to the predicted reduction of the objective function and $0 < \eta_1 < \eta_2 < 1$ are appropriate numbers. It now remains to describe our approximate solution of (2.1).

The choice of the infinity norm for the trust region constraint in problem (2.1) allows us to replace the bound constraints and the trust region constraint in this problem by the bound constraints

$$\max([l]_i\,, [x_k]_i - \Delta_k) \stackrel{\text{def}}{=} [l_k]_i \leq [x_k + s]_i \leq [u_k]_i \stackrel{\text{def}}{=} \min([u]_i\,, [x_k]_i + \Delta_k) \tag{2.3}$$

for $i = 1, \ldots, n$. Problem (2.1) then becomes

$$\begin{array}{ll} \min_{s \in \mathbf{R}^n} & m_k(x_k + s) \\ \text{subject to} & As = 0 \\ & l_k \leq x_k + s \leq u_k. \end{array} \tag{2.4}$$

In order to satisfy the global convergence theory developed in [6], we need to find a feasible point $x_k + s_k$ within the trust region at which the value of the model function is no larger than its value at the Generalized Cauchy Point (GCP). This GCP, denoted x_k^C, is found through a *projected search* on the model along an *approximation* of the projected gradient path (i.e. the projection of the gradient on the feasible set). Note that the determination of the active set (the set of variables that are to be fixed at one of their bounds during the current iteration) takes place when finding the GCP. Since no restriction on the number of variables moving in or out the active set is imposed from one iteration to the other, rapid changes may occur in the active set. This is extremely useful in large-scale optimization problems since the number of iterations required to find the correct active set may hence be considerably smaller than the number of active bounds at the solution. Subsequently we use second order information to refine the GCP and provide a fast ultimate rate of convergence. Therefore, following Murtagh and Saunders [19], we first partition the matrix A as

$$A = \begin{pmatrix} B & S & N \end{pmatrix}$$

where the $m \times m$ submatrix B is nonsingular, and define

$$\{1, \ldots, n\} = \mathcal{B} \cup \mathcal{S} \cup \mathcal{N}, \tag{2.5}$$

the induced partition of the variable indices. For a node-arc incidence matrix, Dantzig [9, p. 356] has shown that the arcs whose indices are in $\mathcal{B}$ form a *spanning tree* of the network. (These arcs are called *basic arcs* while the arcs whose indices are in $\mathcal{S}$ and $\mathcal{N}$ are called *superbasic arcs* and *nonbasic arcs* respectively.) In that case, it is worth using a specialized data structure of the type proposed by Bradley, Brown and Graves [2] that allows us to store and update the basis of the network (i.e. the spanning tree) in a very efficient manner.

According to assumption (AS.8) in [6],

$$\mathcal{A}(x_k^C, l, u) \subseteq \mathcal{A}(x_k + s_k, l, u) \tag{2.6}$$

— the variables of x_k^C that are at a bound must remain fixed when finding a better approximation of a minimizer of (2.4). We then set, at each iteration k,

$$\mathcal{N} = \mathcal{A}(x_k^C, l_k, u_k) \setminus \mathcal{B} \text{ and } \mathcal{S} = \{1, \ldots, n\} \setminus (\mathcal{B} \cup \mathcal{N}). \tag{2.7}$$

Since $j \notin \mathcal{A}(x_k^C, l_k, u_k) \implies j \notin \mathcal{A}(x_k^C, l, u)$, this choice for $\mathcal{N}$ produces a correct set for $\mathcal{S}$ according to assumption (2.6), namely an index set of arcs $\notin \mathcal{B}$ strictly between the bounds l and u. Note that this choice imposes more than the assumption requires, since it further fixes the components of the GCP that are on the trust region boundary (even if they are not at a bound l or u), which seems quite natural.

Using a *reduction matrix* Z as proposed by Murtagh and Saunders [19] (that is a matrix formed by column vectors that span the nullspace of A, yielding the relation $AZ = 0$) and choosing

$$Z = \begin{pmatrix} -B^{-1}S \\ I \\ 0 \end{pmatrix}, \tag{2.8}$$

we then solve approximately problem (2.4) by applying a conjugate gradient algorithm, starting from the GCP, to the equation

$$Z^T H_k Z \, [s]_{\mathcal{S}} = -Z^T g_k.$$

Let $[s_k]_{\mathcal{S}}$ be the approximation found. We define the full trial step s_k by $s_k = ([s_k]_{\mathcal{B}}, [s_k]_{\mathcal{S}}, [s_k]_{\mathcal{N}})$ where $[s_k]_{\mathcal{B}}$ and $[s_k]_{\mathcal{N}}$ satisfy

$$B \, [s_k]_{\mathcal{B}} = -S \, [s_k]_{\mathcal{S}} \tag{2.9}$$

and

$$[s_k]_{\mathcal{N}} = 0.$$

We defer to Section 2.3 the management of the constraints $l_k \le x_k + s \le u_k$ during the conjugate gradient schemes solving problem (2.4). Note that the matrix Z in (2.8) exhibits a useful structure [16]. Indeed, the jth column of Z corresponds to the cycle formed by adding the jth superbasic arc to the spanning tree associated with the basis. This cycle can be decomposed in the jth superbasic arc, joining nodes e and f, say, and its associated *flow augmenting path* (also called *basic equivalent path*), which is the (unique) path between nodes e and f belonging to the tree. Let β_j be the set of indices of the arcs of this path. The element (i, j) of $-B^{-1}S$ is then given by

$$\left[-B^{-1}S\right]_{ij} = \begin{cases} +1 & \text{if } i \in \beta_j \text{ and the } i\text{th basic arc has an orientation iden-} \\ & \text{tical to that of the } j\text{th superbasic arc in the cycle,} \\ -1 & \text{if } i \in \beta_j \text{ and the } i\text{th basic arc has an orientation op-} \\ & \text{posite to that of the } j\text{th superbasic arc in the cycle,} \\ 0 & \text{if } i \notin \beta_j. \end{cases} \tag{2.10}$$

This special structure allows for a compact storage of the matrix Z, as well as for very efficient techniques for computing products that involve this matrix or its transpose (see [22] for more details). Moreover, this last structure is analogous to that of the matrix $-B^{-1}N$ that arises in the computation of the Lagrange multiplier estimates,

$$[g_k]_{\mathcal{N}} - N^T B^{-T} [g_k]_{\mathcal{B}},$$

with the only difference being that the flow augmenting path is now associated with a nonbasic variable instead of a superbasic one.

In order to be sure that assumption (2.6) holds, we further need to impose that the basic arcs whose indices are in $\mathcal{A}(x_k^C, l, u)$ remain fixed when finding the candidate step s_k. But this can be automatically induced by using the concept of *maximal spanning tree*, as introduced by Dembo and Klincewicz in [12], that is a spanning tree which has a maximal number of arcs whose flows are strictly between the bounds l and u (see also [23]). With such a spanning tree, a basic arc whose flow is at a bound is not allowed to belong to the flow augmenting path of a *free* arc (that is an arc whose flow is strictly between its bounds), since otherwise, the replacement of this basic arc with the free one would increase the number of free arcs in the spanning tree, in contradiction with its property of maximality. Given the way the index sets $\mathcal{N}$ and $\mathcal{S}$ are defined in (2.7), every superbasic arc is ensured to be strictly between the bounds l and u, and the use of maximal spanning trees therefore prevents any basic arc that belongs to the flow augmenting path of a superbasic arc to be at one of its bounds. Consequently, since a basic component of s_k computed from (2.9) may be non-zero *only if* its corresponding arc belongs to the flow augmenting path of at least one superbasic arc (see (2.10)), we are sure that the only basic arcs allowed to change during the process are those which are strictly between the bounds l and u. Moreover, using the same argument, we force the basic arcs that are on the trust region boundary to remain fixed by imposing that the spanning tree be maximal *also with respect* to the bounds l_k and u_k (that is to have a maximal number of arcs whose flows are strictly between the bounds l_k and u_k).

Under condition (2.6) and a nondegeneracy condition, the strategy described above is sufficient to ensure that the correct active set is identified after a finite number of iterations (see [6]). We now give, in the next two sections, more details on the computations of the GCP x_k^C and the trial step s_k.

2.2 The Generalized Cauchy Point

Following [6], in order to find a Generalized Cauchy Point, we first need to determine an approximation of a suitable point on the projected gradient path. By this, we mean a feasible point $x_k^C = x_k + s_k^C$ inside the trust region that satisfies the inequality

$$g_k^T s_k^C \leq -\mu_3 \alpha_k(t_k) \tag{2.11}$$

for some fixed $\mu_3 \in (0, 1]$ and $t_k > 0$. Here $\alpha_k(t_k) > 0$ represents the magnitude of the maximum decrease of the linearized model achievable on the intersection of the feasible domain with a box of radius t_k centered at x_k:

$$\begin{array}{rlc} -\alpha_k(t_k) \stackrel{\text{def}}{=} & \min_{d \in \mathbf{R}^n} & g_k^T d \\ & \text{subject to} & Ad = 0 \\ & & l_{t_k} \leq d \leq u_{t_k}, \end{array} \tag{2.12}$$

where

$$[l_{t_k}]_j \stackrel{\rm def}{=} \max([l]_j, [x_k]_j - t_k) - [x_k]_j$$

and

$$[u_{t_k}]_j \stackrel{\rm def}{=} \min([u]_j, [x_k]_j + t_k) - [x_k]_j$$

for $j = 1, \ldots, n$. Furthermore, this point must satisfy the two Goldstein-like conditions

$$m_k(x_k + s_k^C) \leq m_k(x_k) + \mu_1 g_k^T s_k^C \tag{2.13}$$

and

$$\text{either } \; t_k \geq \min[\nu_1 \Delta_k, \nu_2] \; \text{ or } \; m_k(x_k + s_k^C) \geq m_k(x_k) + \mu_2 g_k^T s_k^C, \tag{2.14}$$

where $0 < \mu_1 < \mu_2 < 1$, $\nu_1 \in (0, 1)$ and $\nu_2 \in (0, 1]$ are appropriate constants.

The *GCP Algorithm* given in [6] is a model algorithm for computing a Generalized Cauchy Point that verifies conditions (2.11), (2.13) and (2.14). It is iterative and uses bisection. At each iteration i, given a bisection parameter value $t_i > 0$, it computes first a candidate step s_i that satisfies condition (2.11) (with $t_k = t_i$ and $s_k^C = s_i$), checking then conditions (2.13) and (2.14) (with $t_k = t_i$ and $s_k^C = s_i$), until either an acceptable GCP is found or two candidates $x_k + s_k^l$ and $x_k + s_k^u$ are known that violate condition (2.13) and condition (2.14). Thus, if an acceptable GCP is not yet found, the algorithm carries out a simple bisection linesearch on the model along a particular path between these two points, yielding a suitable GCP in a finite number of iterations. This particular path, called the *restricted path*, is obtained by applying the so-called *restriction operator*,

$$R_{x_k}[y] \stackrel{\rm def}{=} \arg \min_{z \in [x_k, y] \cap X} \|z - y\|_2$$

where $[x_k, y]$ is the segment between x_k and y, on the piecewise linear path consisting of the segment $[x_k + s_k^l, x_k + s_k^p]$ followed by $[x_k + s_k^p, x_k + s_k^u]$, where

$$s_k^p = \max\left[1, \frac{\|s_k^u\|_\infty}{\|s_k^l\|_\infty}\right] s_k^l.$$

This restricted path is an approximation of the unknown projected gradient path between the points $x_k + s_k^l$ and $x_k + s_k^u$ in the sense that each point on this path satisfies condition (2.11) for some $t_k > 0$. It also closely follows the boundary of the feasible domain, as does the projected gradient path. We refer the reader to [6] for a detailed discussion of these concepts.

In order to perform the simple bisection linesearch along the restricted path, a call to the *RS Algorithm* given below is made in the GCP Algorithm. The inner iterations of Algorithm RS are denoted by the index j.

RS Algorithm

Step 0 : initialization. Set $\delta_p = \|s_k^p - s_k^l\|_2$, $\delta_u = \delta_p + \|s_k^u - s_k^p\|_2$, $l_0 = 0$, $u_0 = \delta_u$ and $j = 0$. Then define $\delta_0 = \frac{1}{2}(l_0 + u_0)$.

Step 1 : compute the point on the restricted path corresponding to δ_j.

Step 1.0 : compute the step from x_k to the piecewise linear path. Set

$$s_j \stackrel{\text{def}}{=} \begin{cases} \frac{\delta_j}{\delta_p} s_k^p + (1 - \frac{\delta_j}{\delta_p}) s_k^l & \text{if } \delta_j \leq \delta_p, \\ \\ \frac{\delta_j - \delta_p}{\delta_u - \delta_p} s_k^u + (1 - \frac{\delta_j - \delta_p}{\delta_u - \delta_p}) s_k^p & \text{if } \delta_j \geq \delta_p. \end{cases}$$

Step 1.1 : calculate the smallest value of α such that $x_k + \alpha s_j$ hits a bound. Set

$$\alpha^* = \min \left[\min_{\{i \in \{1,\ldots,n\} | [s_j]_i < 0\}} \left| \frac{[x_k]_i - [l]_i}{[s_j]_i} \right|, \min_{\{i \in \{1,\ldots,n\} | [s_j]_i > 0\}} \left| \frac{[x_k]_i - [u]_i}{[s_j]_i} \right| \right].$$

Step 1.2 : compute the point on the restricted path. Set

$$\alpha_j = \min[1, \alpha^*]$$

and

$$x_j = x_k + \alpha_j s_j.$$

Step 2 : check the stopping conditions. If

$$m_k(x_j) > m_k(x_k) + \mu_1 g_k^T (x_j - x_k), \tag{2.15}$$

then set

$$l_{j+1} = l_j \ \text{ and } \ u_{j+1} = \delta_j,$$

and go to Step 3. Else, if

$$m_k(x_j) < m_k(x_k) + \mu_2 g_k^T (x_j - x_k), \tag{2.16}$$

then set

$$l_{j+1} = \delta_j \ \text{ and } \ u_{j+1} = u_j,$$

and go to Step 3; else (that is if both (2.15) and (2.16) fail), set $x_k^C = x_j$ and STOP.

Step 3 : choose the next parameter value by bisection. Increment j by one, set

$$\delta_j = \frac{1}{2}(l_j + u_j)$$

and go to Step 1.

Note that the point x_j calculated at Step 1 satisfies the constraint $Ax = b$ and minimizes the distance from $x_k + s_j$ in the direction $-s_j$ while satisfying the constraints $l \leq x_j \leq u$, as expected.

As mentioned before, at a given iteration i of the GCP Algorithm we first compute a candidate step s_i that satisfies condition

$$g_k^T s_i \leq -\mu_3 \alpha_k(t_i) \tag{2.17}$$

where t_i is the current bisection parameter value. We obtain s_i by applying a simplex-like algorithm to problem (2.12) (where t_k is replaced by t_i) and by stopping this algorithm as soon as an admissible iterate d_ℓ has been found that verifies

$$|g_k^T d_\ell| \geq \mu_3 \min_{r=1,\ldots,\ell} \left[l_{t_i}^T A^T \pi_r + (u_{t_i} - l_{t_i})^T \mu_r - g_k^T l_{t_i} \right], \tag{2.18}$$

where

$$\pi_r = [g_k]^T_{\mathcal{B}_r} B_r^{-1} \ \text{ and } \ [\mu_r]_j = \max(0, \pi_r A e_j - [g_k]_j) \quad (j = 1, \ldots, n), \tag{2.19}$$

where B_r is the admissible basis associated with some previous candidate d_r, $[g_k]_{\mathcal{B}_r}$ is the basic part of g_k and e_j is the j-th vector of the canonical basis of $\mathbf{R}^n$. Indeed, the right-hand-side of condition (2.18) is an upper bound on the value of $\mu_3 \alpha_k(t_i)$ and (2.18) thus implies condition (2.17) for $s_i = d_\ell$ (see [6] for more details).

Now we give the GCP Algorithm itself. Its inner iterations are denoted by the index i.

GCP Algorithm

Step 0: initialization. Choose $\lambda \in (0,1)$. Set $l_0 = 0$, $u_0 = \Delta_k$, $s_0^l = 0$ and $i = 0$. Also choose s_0^u an arbitrary vector such that $\|s_0^u\|_\infty > \Delta_k$ and an initial parameter $t_0 \in (0, \Delta_k]$.

Step 1: compute a candidate step. Compute a vector s_i such that

$$A s_i = 0 \ \text{ and } \ l_{t_i} \le s_i \le u_{t_i},$$

and

$$g_k^T s_i \le -\mu_3 \alpha_k(t_i).$$

Step 2: check the stopping rules on the model and step. If

$$m_k(x_k + s_i) > m_k(x_k) + \mu_1 g_k^T s_i, \tag{2.20}$$

then set

$$u_{i+1} = t_i \quad s_{i+1}^u = s_i$$

and

$$l_{i+1} = l_i \quad s_{i+1}^l = s_i^l,$$

and go to Step 3. Else, if

$$m_k(x_k + s_i) < m_k(x_k) + \mu_2 g_k^T s_i \tag{2.21}$$

and

$$t_i < \min[\nu_1 \Delta_k, \nu_2], \tag{2.22}$$

then set

$$u_{i+1} = u_i \quad s_{i+1}^u = s_i^u$$

and

$$l_{i+1} = t_i \quad s_{i+1}^l = s_i,$$

and go to Step 3. Else (that is if (2.20) and either (2.21) or (2.22) fail), then set

$$x_k^C = x_k + s_i$$

and STOP.

Step 3: define a new trial step by bisection. We distinguish two mutually exclusive cases.

Case 1: $s^l_{i+1} = s^l_0$ or $s^u_{i+1} = s^u_0$. Set

$$t_{i+1} = \lambda(l_{i+1} + u_{i+1}),$$

increment i by one and go to Step 1.

Case 2: $s^l_{i+1} \neq s^l_0$ and $s^u_{i+1} \neq s^u_0$. Set

$$s^l_k = s^l_{i+1} \text{ and } s^u_k = s^u_{i+1},$$

define

$$s^p_k = \max\left[1, \frac{\|s^u_k\|_\infty}{\|s^l_k\|_\infty}\right] s^l_k,$$

apply the RS Algorithm to find a GCP x^C_k and STOP.

For the computation of s_i in Step 1, we have implemented a self-contained routine that uses the same data structure as that representing problem (1.1) and is a particular implementation of the simplex algorithm specialized to network problems, along the lines described in [2], [16] and [17]. This routine includes at each iteration the computation of the vectors π_r and μ_r from (2.19) as well as the update of the upper bound on the value of $\mu_3\alpha_k(t_i)$ given in (2.18), and stops as soon as an appropriate inexact solution is computed. This implementation provides in particular a *total pricing* routine (see [17]) for seeking a nonbasic candidate to enter the basis, since the vector μ_r has to be totally evaluated at each iteration. In order to compare the performances of this last algorithm with one that completely solves problem (2.12) (as required if μ_3 is set to 1), we have also implemented a routine that finds the exact solution of (2.12), without adding the extra burden of computing the quantities required for an approximate solution (namely μ_r and the upper bound on $\mu_3\alpha_k(t_i)$), but rather using a *partial pricing* routine to select a nonbasic arc to be moved. More precisely, we select sets of thirty variables taken at regular intervals among the nonbasic variables and test each variable in the successive sets until a candidate to enter the basis is found.

We have left unspecified the parameter $\lambda \in (0,1)$ in the GCP Algorithm (see Case 1 of Step 3) in order to test the effect of varying its value. Indeed, in order to avoid an excessive number of computations of a candidate step s_i in Step 1 — the most costly calculation of the algorithm — it could be worth to accelerate the branching to the second case of Step 3 by choosing a smaller value for λ than the classical 0.5.

The above algorithm for the calculation of a GCP has the advantage of avoiding the repeated computation of the projection on the feasible domain, which is a *quadratic* program. Instead we repeatedly compute an approximate solution of *linear* programs. This can be related to the convex combination algorithm originally suggested by Frank and Wolfe (see [20]) for solving quadratic programming problems with linear constraints. The Frank and Wolfe algorithm is based on finding a descent direction by minimizing a linear approximation to the function subject to the linear constraints. A linesearch on the quadratic objective function along the descent direction found is then performed to determine the next iterate.

2.3 The candidate step s_k

In this section, we analyse two strategies for solving problem (2.4), or more precisely, for finding an approximate solution to the reduced equation

$$Z^T H_k Z [s]_{\mathcal{S}} = -Z^T g_k. \tag{2.23}$$

These strategies both use a *truncated conjugate gradient* technique, starting from the GCP, but differ in the way they handle the bound constraints

$$l_k \leq x_k + s \leq u_k \tag{2.24}$$

during the conjugate gradient iteration.

The conjugate gradient method is well suited to solve (2.23) without forming the reduced Hessian $Z^T H_k Z$ (which may be considerably denser than both Z and H_k), since it only requires matrix-vector products of the form $Z^T H_k Z v$. These products can be computed relatively cheaply by forming, in turn, $v_1 = Zv$, $v_2 = H_k v_1$ and $v_3 = Z^T v_2$. This is all the cheaper here as a sparse Hessian H_k and a sparse representation of Z can be stored, due to the partially separable structure of the objective f and the structure of the matrix Z (see (2.8) and (2.10)).

Both strategies terminate the conjugate gradient iteration in the solution of (2.23) at the point $x = x_k + s$ whenever:

- The *reduced residual norm* at x (i.e. the norm of the reduced gradient of the model at the point x) is small enough, that is

  $$\|Z^T H_k Z [s]_{\mathcal{S}} + Z^T g_k\|_2 \leq \eta_k,$$

  where

  $$\eta_k = \max\left[\sqrt{\epsilon_M}, \min[0.01, \|Z^T g_k\|_2]\right] \|Z^T g_k\|_2 \tag{2.25}$$

  and ϵ_M is the relative machine precision. This stopping rule allows for better and better approximations to the solution of the Newton equation (2.23) when close to a local minimizer of problem (1.1) and is the essence of a *truncated* Newton scheme.

- $\mathcal{S} = \emptyset$, i.e. there is no way to better refine the current solution x.

- A direction of negative curvature has been encountered.

- An excessive number of iterations has been taken.

The main characteristics of the proposed strategies are the following:

- At each recurrence of a conjugate gradient iteration, Strategy 1 will verify if feasibility with respect to the bound constraints (2.24) is still respected. In the case where a bound is reached, the conjugate gradient iteration is temporarily stopped and the current maximal spanning tree is possibly updated, depending on the type of bound encountered. Thereafter, the active set is updated according to the decomposition (2.5), where the index set $\mathcal{S}$ corresponds to the arcs whose current flow is strictly between the bounds l_k and u_k. The conjugate gradient iteration is then possibly restarted.

- In Strategy 2, an initial active set is computed with respect to the constraints active at the GCP, as in Strategy 1, but this active set is not altered during the conjugate gradient iteration. Thus one complete iteration is carried out without taking into account the bound constraints (2.24) until the level accuracy η_k on the reduced residual norm is reached, with the exception of encountering negative curvature. This avoids the necessity to update the current maximal spanning tree as well as the current active set since there is no need to stop and restart the conjugate gradient iteration. At the end of the algorithm, a feasible iterate with respect to the bound constraints (2.24) is calculated by backtracking. Basically, this scheme consists of making a step of maximal length from the GCP to the bound constraints (2.24) in an approximate Newton direction, which is computed in the subspace of the superbasic variables defined at the GCP.

The principal motivation behind these strategies is to observe the behaviour of the method when varying the way the bound constraints are handled. Indeed, we allow the search to go farther (also reducing the amount of work) when not taking bound constraints into account during the conjugate gradient iteration, but with the additional danger that the decrease obtained is less after the backtracking has been performed.

Now we specify the two strategies in more detail. In the descriptions given below, we denote by r the *residual* vector $-(g_k + H_k s)$. For a given vector v and a given partition $\mathcal{B} \cup \mathcal{S} \cup \mathcal{N}$ of the set $\{1, \ldots, n\}$, we also define the corresponding *reduced* vector v^r as the vector of $\mathbf{R}^n$ defined componentwise by

$$[v^r]_i \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } i \in \mathcal{B} \cup \mathcal{N} \\ \left[Z^T v\right]_j & \text{if } i \text{ is the } j\text{th element of } \mathcal{S} \text{ (that is the } j\text{th superbasic arc).} \end{cases} \tag{2.26}$$

The first strategy is thus mainly based on the idea of maintaining feasibility of the iterates (with respect to the bound constraints (2.24)) during all the conjugate gradient process, while allowing to restart it when a bound is encountered.

Strategy 1

Step 0 : initialization. The GCP, $x_k^C = x_k + s_k^C$, and an initial maximal spanning tree whose indices define the set $\mathcal{B}$ are given. Set $x = x_k^C$ and $r = -(g_k + H_k s_k^C)$.

Step 1 : the conjugate gradient iteration.

As long as there exist arcs $\notin \mathcal{B}$ which are strictly between the bounds l_k and u_k, we continue the conjugate gradient iteration to further minimize the reduced model of the objective function at the point x_k. Each time a restarting is considered, we redefine the index set $\mathcal{S}$ and equation (2.23) accordingly, and solve this last equation starting from the current point x, until one of the stopping rules mentioned above is satisfied or a bound is encountered.

Step 1.0 : define the active set. Set

$$\mathcal{N} = \mathcal{A}(x, l_k, u_k) \setminus \mathcal{B}$$

and deduce $\mathcal{S}$ from the partition (2.5).

If $\mathcal{S} = \emptyset$, go to Step 2. Otherwise, compute the matrices B, S, N and Z from the partition (2.5) and (2.8).

Step 1.1 : restart the conjugate gradient iteration.

Now that the subspace where the minimization can take place is fixed (namely the space spanned by the superbasic variables indexed by $\mathcal{S}$), we can proceed with the conjugate gradient iteration.

Step 1.1.0 : initialization before restarting. Compute the *reduced residual* r^r from (2.26) and the relative accuracy level η_k from equation (2.25).
Set $d = 0$, $\beta = 0$ and $\rho_2 = \|r^r\|_2^2$.

Step 1.1.1 : test for the required accuracy.
If $\rho_2 \leq \eta_k^2$, go to Step 2.

Step 1.1.2 : conjugate gradient recurrences. Compute

$$[d]_{\mathcal{S}} = [r^r]_{\mathcal{S}} + \beta \, [d]_{\mathcal{S}} \, ,$$

$[d]_{\mathcal{B}}$ from

$$B \, [d]_{\mathcal{B}} = -S \, [d]_{\mathcal{S}}$$

and set $d = ([d]_{\mathcal{B}} \, , [d]_{\mathcal{S}} \, , 0)$.
Compute the vectors $y = H_k d$, y^r from (2.26), and the curvature $\gamma = d^T y$.
Find α_1, the largest value of α for which $l_k \leq x + \alpha d \leq u_k$.
If $\gamma \leq 0$, then set

$$x = x + \alpha_1 d$$

and go to Step 1.2. Otherwise, calculate $\alpha_2 = \rho_2 / \gamma$.
If $\alpha_2 \geq \alpha_1$, then set

$$x = x + \alpha_1 d$$

$$r = r - \alpha_1 y$$

and go to Step 1.2. Otherwise, set

$$x = x + \alpha_2 d$$

$$r = r - \alpha_2 y$$

$$r^r = r^r - \alpha_2 y^r$$

$$\rho_1 = \rho_2$$

$$\rho_2 = \|r^r\|_2^2$$

$$\beta = \frac{\rho_2}{\rho_1}$$

and go to Step 1.1.1.

Step 1.2 : update the maximal spanning tree. Update the index set $\mathcal{B}$ in order to keep a maximal spanning tree (see further on).

If $\gamma \leq 0$, go to Step 2. Otherwise, go to Step 1.

Step 2 : termination of the conjugate gradient iteration. Set

$$s_k = x - x_k.$$

The second strategy given below totally ignores the bound constraints (2.24) during the conjugate gradient process and solves equation (2.23) for the active set defined by the constraints active at the GCP.

Strategy 2

Step 0 : initialization. The GCP, $x_k^C = x_k + s_k^C$, and an initial maximal spanning tree whose indices define the set $\mathcal{B}$ are given. Set $x = x_k^C$ and $r = -(g_k + H_k s_k^C)$.

Initialize the active set, that is set

$$\mathcal{N} = \mathcal{A}(x_k^C, l_k, u_k) \setminus \mathcal{B},$$

and deduce $\mathcal{S}$ from the partition (2.5).

If $\mathcal{S} = \emptyset$, go to Step 4. Otherwise, compute the matrices B, S, N and Z from the partition (2.5) and (2.8).

Finally, compute the reduced residual r^r from (2.26) and the relative accuracy level η_k from equation (2.25). Set $d = 0$, $\beta = 0$ and $\rho_2 = \|r^r\|_2^2$.

Step 1 : the conjugate gradient iteration.

Given $\mathcal{S}$, we solve equation (2.23) starting from x_k^C until one of the stopping rules mentioned above is satisfied.

Step 1.0 : test for the required accuracy.

If $\rho_2 \leq \eta_k^2$, go to Step 2.

Step 1.1 : conjugate gradient recurrences. Compute

$$[d]_{\mathcal{S}} = [r^r]_{\mathcal{S}} + \beta\, [d]_{\mathcal{S}},$$

$[d]_{\mathcal{B}}$ from

$$B\,[d]_{\mathcal{B}} = -S\,[d]_{\mathcal{S}}$$

and set $d = ([d]_{\mathcal{B}}, [d]_{\mathcal{S}}, 0)$.

Compute the vectors $y = H_k d$, y^r from (2.26), and the curvature $\gamma = d^T y$.

Calculate $\alpha_2 = \rho_2/\gamma$.

If $\gamma \leq 0$, then set

$$\alpha_2 = -\alpha_2$$

$$[x]_{\mathcal{S}} = [x]_{\mathcal{S}} + \alpha_2 [d]_{\mathcal{S}}$$

and go to Step 2. Otherwise, set

$$[x]_{\mathcal{S}} = [x]_{\mathcal{S}} + \alpha_2 [d]_{\mathcal{S}}$$

$$r^r = r^r - \alpha_2 y^r$$

$$\rho_1 = \rho_2$$

$$\rho_2 = \|r^r\|_2^2$$

$$\beta = \frac{\rho_2}{\rho_1}$$

and go to Step 1.0.

Step 2 : find a feasible iterate. Compute

$$[s]_{\mathcal{S}} = [x]_{\mathcal{S}} - \left[x_k^C\right]_{\mathcal{S}},$$

$[s]_{\mathcal{B}}$ from

$$B\,[s]_{\mathcal{B}} = -S\,[s]_{\mathcal{S}}$$

and set $s = ([s]_{\mathcal{B}}, [s]_{\mathcal{S}}, 0)$.

Find α^*, the largest value of α for which $l_k \leq x_k^C + \alpha s \leq u_k$.

If $\gamma > 0$, then set

$$\alpha^* = \min[1, \alpha^*].$$

Set

$$x = x_k^C + \alpha^* s.$$

Step 3 : update the maximal spanning tree. Update the index set $\mathcal{B}$ in order to keep a maximal spanning tree (see further on).

Step 4 : termination of the conjugate gradient iteration. Set

$$s_k = x - x_k.$$

In this last strategy, the amount of work decreases mainly in terms of pivoting steps and in terms of restarting steps. However, since the bound constraints (2.24) are not taken into account, we may fear that Strategy 2 is not adapted to problems that have many active bounds at the solution.

In order to maintain a maximal spanning tree when a pivoting step is required, we need to take into account the bound constraints of both problem (1.1) and problem (2.4) (the latter varying from one iteration to the other, depending on the trust region size), since the spanning tree has to remain maximal with respect to the bounds l and u *and* l_k and u_k. As illustrated by the following example, it is not sufficient to only consider the maximality of the spanning tree with respect to the bounds l_k and u_k. Suppose indeed, when moving to a point x, that the basic arc of index i hits a bound such that not only $i \in \mathcal{A}(x, l_k, u_k)$, but also $i \in \mathcal{A}(x, l, u)$, and that no arc $j \notin \mathcal{B}$ satisfying $[l_k]_j < [x]_j < [u_k]_j$ may be found to pivot with. In that case, if the current maximal spanning tree remains unmodified and if there exists an arc of index j, say, such that $j \in \mathcal{A}(x, l_k, u_k)$, $i \in \beta_j$ (i.e. arc i belongs to the flow augmenting path of arc j), but $j \notin \mathcal{A}(x, l, u)$, this spanning tree will not be maximal any more with respect to the bounds l and u as soon as the trust region constraint vanishes from (2.4) or is modified. Therefore, based on the observation that $i \in \mathcal{A}(x, l, u) \Longrightarrow i \in \mathcal{A}(x, l_k, u_k)$ and $j \notin \mathcal{A}(x, l_k, u_k) \Longrightarrow j \notin \mathcal{A}(x, l, u)$, we consider the following algorithm for maintaining a maximal spanning tree:

> If, for some $i \in \mathcal{B}$,
>
> $$i \in \mathcal{A}(x, l, u)$$
>
> or
>
> $$i \notin \mathcal{A}(x, l, u) \text{ and } i \in \mathcal{A}(x, l_k, u_k),$$
>
> then determine (if possible) $j \notin \mathcal{B}$ such that $i \in \beta_j$, $\min \left[\left| [x]_j - [l_k]_j \right|, \left| [x]_j - [u_k]_j \right| \right]$ is maximum and either
>
> $$j \notin \mathcal{A}(x, l, u)$$
>
> or
>
> $$j \notin \mathcal{A}(x, l_k, u_k),$$
>
> respectively. Then redefine the set $\mathcal{B}$ by
>
> $$\mathcal{B} = \mathcal{B} \setminus \{i\} \cup \{j\}$$
>
> and update the submatrix B accordingly, performing a pivoting step as described in [2].

Note that the choice of j in the above description is intended to allow larger steps in the next search, which may result in a more useful decrease of the cost function (see [21]).

2.4 Optimality test

We consider that optimality for problem (1.1) is reached whenever the objective function cannot be further reduced at the current iterate x_k. This may be checked in the following manner.

- Select the arcs that allow for a possible improvement. This amounts to finding the arcs $\notin \mathcal{B}$ which are either strictly between the bounds l and u or at one of these bounds, but whose release may induce a decrease in the objective function. (These last arcs are found through an examination of the corresponding Lagrange multipliers.)

- Remove the so-called *blocked* arcs ([11]), that is the arcs at a bound l or u whose release causes the immediate violation of another bound l or u for one of the arcs of their flow augmenting path. This may be easily verified using the following test:

If arc j is such that, either

$$[x_k]_j = [u]_j \text{ and } \exists i \in \beta_j \text{ such that}$$

$$\left([-B^{-1}N]_{ij} = 1 \text{ and } [x_k]_i = [l]_i\right) \text{ or } \left([-B^{-1}N]_{ij} = -1 \text{ and } [x_k]_i = [u]_i\right),$$

or

$$[x_k]_j = [l]_j \text{ and } \exists i \in \beta_j \text{ such that}$$

$$\left([-B^{-1}N]_{ij} = 1 \text{ and } [x_k]_i = [u]_i\right) \text{ or } \left([-B^{-1}N]_{ij} = -1 \text{ and } [x_k]_i = [l]_i\right),$$

then it is blocked.

(Note that this situation cannot occur for the arcs that are strictly between the bounds l and u, because of the properties of the maximal spanning tree.)

- Denoting by $\mathcal{S}$ the set of indices obtained from the above selection, deduce the set $\mathcal{N}$ from the partition (2.5) and define an active set accordingly.
- Check if the current iterate x_k is optimal on this active set, that is, if the corresponding reduced gradient at x_k is null.

This framework may be summarized by the following algorithm.

OT Algorithm

Step 0 : $\mathcal{B}$ is given. Set $\mathcal{S} = \emptyset$ and $\mathcal{N} = \{1, \ldots, n\} \setminus \mathcal{B}$.

Step 1 : For each $j \in \{1, \ldots, n\} \setminus \mathcal{B}$, redefine $\mathcal{S}$ and $\mathcal{N}$ in the following way:

If

$$j \notin \mathcal{A}(x_k, l, u),$$

then redefine $\mathcal{S}$ and $\mathcal{N}$ by

$$\mathcal{S} = \mathcal{S} \cup \{j\} \text{ and } \mathcal{N} = \mathcal{N} \setminus \{j\}.$$

Otherwise, compute the Lagrange multiplier estimate associated with the jth variable, namely

$$[\sigma]_j = [g_k]_j + \sum_{i \in \beta_j} \left[-B^{-1}N\right]_{ij} [g_k]_i \, .$$

If $[x_k]_j$ is not potentially blocked (see above), and if either

$$[\sigma]_j < 0 \text{ and } [x_k]_j = [l]_j$$

or

$$[\sigma]_j > 0 \text{ and } [x_k]_j = [u]_j \, ,$$

then redefine $\mathcal{S}$ and $\mathcal{N}$ by

$$\mathcal{S} = \mathcal{S} \cup \{j\} \text{ and } \mathcal{N} = \mathcal{N} \setminus \{j\}.$$

Step 2 : Compute the matrices B, S, N and Z from the partition (2.5) and (2.8).

If $\mathcal{S} = \emptyset$ or
$$\|Z^T g_k\|_\infty \le \eta_3,$$
STOP (x_k is a local optimum within the required accuracy).

The constant η_3 whose choice controls the final accuracy requirement will be specified later.

2.5 The specific algorithm

We are now in position to specify our trust region algorithm for nonlinear network optimization in its entirety.

TRNNO Algorithm

Step 0 : The bounds l and u, the vector b and the network associated with the matrix A are given.

Compute a feasible starting point x_0 (if not given) and an initial trust region radius Δ_0.

Compute $f(x_0)$, g_0 and H_0.

Find an initial maximal spanning tree of the network, defining a set of basic indices $\mathcal{B}$.

Set $k = 0$.

Step 1 : Given η_3, test the optimality of the current iterate x_k using the OT Algorithm of Section 2.4 and STOP if x_k is optimal.

Step 2 : Calculate the bounds l_k and u_k from equation (2.3). Given μ_3, find a Generalized Cauchy Point x_k^C using the GCP Algorithm detailed in Section 2.2. (Also include an updating phase for the maximal spanning tree.)

Step 3 : Compute the active set $\mathcal{A}(x_k^C, l_k, u_k)$ and apply one of the two strategies proposed in Section 2.3, using a truncated conjugate gradient scheme, to find an approximation $x_k + s_k$ to the minimizer of the trust region problem (2.4), with the additional restriction that the variables whose indices are in $\mathcal{A}(x_k^C, l_k, u_k)$ remain fixed at the corresponding values of x_k^C. (Also include an updating phase for the maximal spanning tree, according to the strategy used.)

Step 4 : Compute $f(x_k + s_k)$ and
$$\rho_k = \frac{f(x_k) - f(x_k + s_k)}{m_k(x_k) - m_k(x_k + s_k)}.$$

Step 5 : If $\rho_k > \eta_1$, then set
$$x_{k+1} = x_k + s_k$$
and update g_{k+1} and H_{k+1} accordingly. Otherwise, set
$$x_{k+1} = x_k$$

and update the maximal spanning tree (with respect to the bounds l and u only).

Set Δ_{k+1} according to equation (2.2), increment k by one and go to Step 1.

3 Numerical experiments

In this section, we analyse and compare the various versions of our algorithm and we also briefly interpret our results when varying the storage scheme, the conditioning, the dimension and the nonlinearity of the problem, the final accuracy level (η_3) and the type of bounds imposed on the variables, as in [21] and [22]. We then consider our algorithm in comparison with the LSNNO[1] routine, developed by Toint and Tuyttens in [21]–[22], that uses a linesearch approach rather than a trust region approach to solve problem (1.1). Although it might have been instructive to compare the present algorithm with another such as GENOS [1] and an interior point method like [3], the amount of additional work would have been prohibitive and we preferred to use an algorithm recognized to be competitive for which we had direct access to both the authors and the software.

We have experimented on all the test problems of [21] for which the first and second derivatives were available. Indeed, though the framework presented here is well suited to large dimensional problems and can be used in conjunction with partitioned secant updating techniques on the general class of partially separable problems (see [14] and [15]), the purpose of this paper is to show the viability of the framework proposed and studied in [6], as well as its efficiency on large-scale nonlinear problems. Consequently, the results are presented for problems with easily computable first and second derivatives. For the same reason, we did not consider any preconditioning in our present implementation.

We have mainly tested problems obtained by varying the five parameters of the so-called model test problem $P(\ell, a, c, i, r)$ constructed by Toint and Tuyttens [21], where

ℓ defines the number of arcs $n = 2(2\ell+1)(2\ell+2)$ and the number of nodes $n_n = (2\ell+2)^2$ of the problem,

a defines the nonlinearity of the function (for $a = 0$ the function is a simple quadratic),

c is an estimate of the condition number of the objective's Hessian matrix projected in the subspace of variables that satisfy the network constraints,

i and r determine a specific set of bounds on the flows (for $i = 0$ no bounds are imposed, for $i = 1$ a lower bound equal to r is imposed on the flows whose index is a multiple of three, for $i = -1$ some flows are fixed while others are bounded, principally those on the border of the grid with lower bound equal to r).

A brief description of this model test problem follows, the reader being refered to [21] for more details. The network is constructed as a square planar grid. An example with $\ell = 2$ is shown in

[1] LSNNO is available from NETLIB.

Figure 1. The supply/demand vector is

$$b_1 = +10, \quad b_j = 0 \; (j = 2, \dots, n_n - 1), \quad b_{n_n} = -10.$$

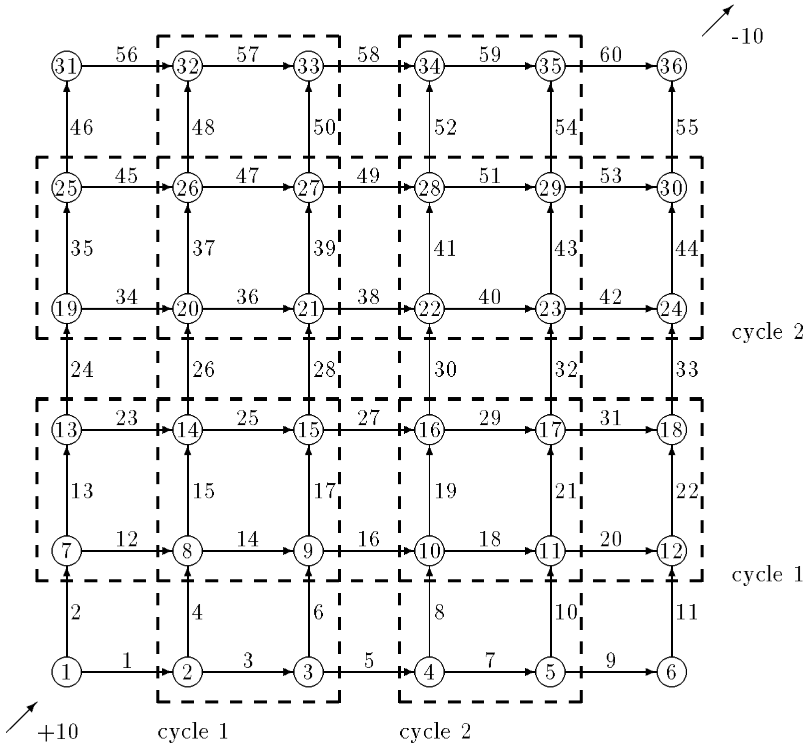

Figure 1: The network of the model test problem for $\ell = 2$

Furthermore, sets of ℓ horizontal cycles and ℓ vertical cycles are distinguished in the grid (see the dashed lines in the example of Figure 1). We define, for $i = 1, \dots, n$,

$$j(i) \stackrel{\text{def}}{=} \begin{cases} s & \text{if the } i\text{th arc belongs to cycle } s \text{ (horizontal or vertical)}, \\ 0 & \text{if the } i\text{th arc does not belong to any cycle.} \end{cases}$$

The objective function is then given by

$$f(x) = \frac{1}{100} \sum_{i=1}^{n} \alpha_i [x]_i^2 + \frac{a}{100} \left(\sum_{i=1}^{n-1} \sqrt{1 + [x]_i^2 + ([x]_i - [x]_{i+1})^2} + \frac{1}{1200} \left[10 + \sum_{i=1}^{n} (-1)^i [x]_i \right]^4 \right). \quad (3.1)$$

where

$$\alpha_i \stackrel{\text{def}}{=} \begin{cases} 10^{\frac{j(i)-1}{\ell-1} \log_{10} c} & \text{if } j(i) \geq 1, \\ 1 & \text{if } j(i) = 0. \end{cases}$$

We have also tested the so-called Dembo's test problems given by Dembo in [11]. These problems are summarized in Table 1 (where n denotes the number of arcs and n_n denotes the number of nodes). All of them are *totally* separable, convex and rather ill-conditioned (the condition number of the reduced Hessian at the solution varying between 10^4 and 10^8), and the number of bounds active at the solution is small, compared to n.

Name	n	n_n	Description
W30	46	30	Small Dallas water distribution model
W150	196	150	Medium Dallas water distribution model
W666	906	666	Large Dallas water distribution model
MB64	117	64	Small Thai matrix balancing problem
MB1116	2230	1116	Large Thai matrix balancing problem

Table 1: Dembo's test problems

All the computations have been performed in double precision on a DEC VAX 3500, under VMS, using the standard Fortran Compiler ($\epsilon_M \simeq 1.39 \times 10^{-17}$).

The tests reported below all use the following values for the algorithm's constants (suggested in [6]):

$$\eta_1 = 0.25 \text{ and } \eta_2 = 0.75, \qquad \mu_1 = 0.1 \text{ and } \mu_2 = 0.9, \qquad \nu_1 = 10^{-5} \text{ and } \nu_2 = 0.01.$$

In order to allow the initial parameter t_0 in the GCP Algorithm to be more refined than Δ_k itself (since this last value represents a trust region radius for the quadratic model much more than for the linear model used in Step 1), we have selected the following value,

$$t_0 = \min\left[\left|\frac{g_k^T g_k}{g_k^T H_k g_k}\right| \|g_k\|_\infty, \Delta_k\right], \tag{3.2}$$

where the first quantity in brackets is the distance from x_k to the minimum of the quadratic model in the steepest descent direction, computed in the infinity norm. The value of μ_3 in the GCP Algorithm (that can be interpreted as the level of solution of the linear network problem (2.12)) is specified for each table of results given below. We have chosen the value 0.1 (rather than the classical value 0.5) for the scalar λ in the GCP Algorithm. This is intended to speed up the branching to the second case of Step 3 in this algorithm, therefore possibly reducing the number of times a candidate step s_i is computed in Step 1, since this last calculation is expected to be expensive compared with the rest of the algorithm. The final accuracy level η_3 in the OT Algorithm is specified for each model problem and is set to 10^{-2} for the Dembo's test problems, as recommended in [11]. In all cases, the (possibly infeasible) starting point is the origin. A feasible starting point x_0 as required in the statement of the TRNNO Algorithm is then computed via an "all artificial start Phase 1" (see [21]). This allows comparison with the LSNNO routine that starts with the same point. (Note that since the cpu-time for the computation of this point, when required, is always negligible compared with the overall cpu-time, it will be ignored in the cpu-times given in the tables.) Finally, the initial trust region radius is fixed to the following value in our tests:

$$\Delta_0 = \min[\|g_0\|_2, 100]. \tag{3.3}$$

3.1 Comparison between the different versions

In this section we comment on the five Dembo's test problems and on twenty others selected from particular choices in [21]–[22] of the model test problem's parameters. Table 2 reports the characteristics of these twenty test problems which are divided into six subsets, according to the different features tested in [21]–[22] and mentioned above. Note that the symbols I, E and O in the last column of this table are used to denote storage using internal dimensions (I), elemental dimensions (E) or using one element (O) for the Hessian matrix (see [21]–[22] for more details). When required for comparisons in the next section, we report the relevant numerical results of all the twenty tests, even if some of these are identical (namely MP6, MP11, MP16 and MP19).

	Name	ℓ	a	c	i	r	η_3	storage
	MP1	8	1	10^3	-1	$\frac{1}{10}$	10^{-5}	I
Storage scheme	MP2	8	1	10^3	-1	$\frac{1}{10}$	10^{-5}	E
	MP3	8	1	10^3	-1	$\frac{1}{10}$	10^{-5}	O
	MP4	8	1	1	1	$\frac{1}{10}$	10^{-5}	I
	MP5	8	1	10	1	$\frac{1}{10}$	10^{-5}	I
Conditioning	MP6	8	1	10^2	1	$\frac{1}{10}$	10^{-5}	I
	MP7	8	1	10^3	1	$\frac{1}{10}$	10^{-5}	I
	MP8	8	1	10^4	1	$\frac{1}{10}$	10^{-5}	I
	MP9	8	1	10^5	1	$\frac{1}{10}$	10^{-5}	I
	MP10	4	1	10^2	1	$\frac{1}{10}$	10^{-5}	I
Dimension	MP11	8	1	10^2	1	$\frac{1}{10}$	10^{-5}	I
	MP12	12	1	10^2	1	$\frac{1}{10}$	10^{-5}	I
Nonlinearity	MP13	8	0	10^2	1	$\frac{1}{10}$	10^{-5}	I
	MP14	12	0	10^2	-1	$\frac{1}{10}$	10^{-5}	I
	MP15	8	1	10^2	1	$\frac{1}{10}$	10^{-3}	I
Final accuracy	MP16	8	1	10^2	1	$\frac{1}{10}$	10^{-5}	I
	MP17	8	1	10^2	1	$\frac{1}{10}$	10^{-7}	I
	MP18	8	1	10^2	0	0	10^{-5}	I
Type of bounds	MP19	8	1	10^2	1	$\frac{1}{10}$	10^{-5}	I
	MP20	8	1	10^2	1	$\frac{1}{5}$	10^{-5}	I

Table 2: The model test problems

We introduce the notation used in the tables presenting the results:

it: the number of major iterations (in the TRNNO Algorithm of Section 2.5);

gcp: the total number of iterations in the GCP calculations;

avn: the average number of GCP calculations per major iteration;

cg: the total number of conjugate gradient recurrences;

nf: the number of function evaluations (i.e. the number of element function evaluations divided by the number of elements);

ng: the number of gradient evaluations (i.e. the number of element gradient evaluations divided by the number of elements);

nH: the number of Hessian evaluations (i.e. the number of element Hessian evaluations divided by the number of elements);

np: the number of maximal spanning tree updates where a pivoting step occurs;

gcpcpu: the cpu-time in seconds for the GCP calculations;

cgcpu: the cpu-time in seconds for the conjugate gradient recurrences;

totcpu: the total cpu-time in seconds (Phase 1 excluded);

ST1: Strategy 1 for computing a candidate step s_k;

ST2: Strategy 2 for computing a candidate step s_k.

Note that fractional numbers of function, gradient or Hessian evaluations are expected, since the partial separability of the objective allows skipping the re-evaluation of the elements whose variables have not been modified since the last evaluation. On the other hand, for the sake of clarity, we round off the cpu-times to the nearest integer number.

We first turn our attention to the computation of a candidate step s_i at Step 1 of the GCP Algorithm. As already mentioned in Section 2.2, we have implemented a *total pricing* routine that *approximately* solves problem (2.12). We have tested this routine for different values of μ_3. The results presented in Table 3 for a representative sample of the twenty-five problems have been obtained when using Strategy 1 for the computation of a candidate step s_k.

Problem	μ_3	it	gcp	avn	cg	gcpcpu	cgcpu	totcpu
MP4	1	11	31	2.8	587	118	110	241
	0.9	11	31	2.8	564	103	106	223
	0.6	11	31	2.8	530	78	99	190
MP8	1	7	30	4.3	2433	60	441	510
	0.9	9	43	4.8	3315	63	607	681
	0.6	11	47	4.3	3475	31	619	663
MP12	1	7	28	4.0	1078	333	479	833
	0.9	8	34	4.2	1254	280	554	857
	0.6	9	31	3.4	1125	88	489	602
MP14	1	5	16	3.2	291	30	29	68
	0.9	6	24	4.0	357	20	35	66
	0.6	13	39	3.0	603	12	58	88
MP16	1	8	30	3.7	815	83	152	244
	0.9	8	29	3.6	871	70	164	243
	0.6	8	27	3.4	726	49	134	193
MP18	1	12	45	3.7	823	113	166	295
	0.9	12	45	3.7	810	109	164	288
	0.6	12	45	3.7	827	78	166	260
W150	1	15	61	4.1	362	6	5	16
	0.9	15	61	4.1	405	5	6	14
	0.6	18	73	4.1	442	3	6	14
W666	1	19	97	5.1	1314	247	110	385
	0.9	21	108	5.1	1890	146	159	334
	0.6	26	130	5.0	2076	64	162	260
MB1116	1	39	60	1.6	15039	2834	4632	7541
	0.9	40	57	1.4	13806	1886	3643	5599
	0.6	40	57	1.4	14587	1481	4739	6302

Table 3: Strategy 1, total pricing, $\mu_3 = 1$, 0.9 and 0.6

We first observe that the number of major iterations usually increases when the value of μ_3 decreases (especially for $\mu_3 = 0.6$). The reason is that for smaller and smaller values of μ_3, the GCP is allowed to be chosen further and further from the projected gradient path. This exhibits the importance of the part played by the GCP in our class of trust region methods and the need of computing a sufficiently good approximation of this point on the projected gradient path. The total number of GCP iterations increases accordingly. However, we observe that the average number of GCP calculations per major iteration decreases with the value of μ_3, while the cpu-times for the GCP calculations considerably decrease, particularly for larger problems (such as MP12, MP14, W666 and MB1116). This is due to the fact that the solution of the linear network problem (2.12) may be stopped prematurely when finding an approximate solution. Nevertheless, comparing the total cpu-times, we conclude that it is worthwhile solving (2.12) approximately whenever the GCP found does not depart too much from the projected gradient path and the total number of iterations is largely unaffected (see MP4, MP16 and MP18). This means that the value of μ_3 must be reduced with care.

Problem	it	nf	ng	nH	np	gcp	cg	gcpcpu	cgcpu	totcpu
MP1	6	5.9	5.9	5.0	0	27	393	8	63	79
MP2	6	5.9	5.9	5.0	0	27	402	53	628	709
MP3	6	7.0	7.0	6.0	0	27	400	50	571	649
MP4	11	11.3	11.3	10.4	6	31	583	22	111	147
MP5	9	9.5	9.5	8.5	4	29	683	18	133	163
MP6	8	8.5	8.5	7.5	5	30	816	17	154	182
MP7	8	8.5	8.5	7.5	5	36	1610	19	306	334
MP8	7	7.5	7.5	6.6	5	31	2450	15	453	477
MP9	10	10.3	10.3	9.3	3	50	6649	24	1260	1297
MP10	5	5.6	5.6	4.7	3	17	190	2	10	14
MP11	8	8.5	8.5	7.5	5	30	816	17	154	182
MP12	7	7.5	7.5	6.6	54	28	1116	39	492	551
MP13	5	5.4	5.4	4.5	7	19	521	7	37	49
MP14	5	3.7	3.7	3.1	3	16	295	6	29	44
MP15	7	7.5	7.5	6.5	5	23	713	14	134	157
MP16	8	8.5	8.5	7.5	5	30	816	17	155	182
MP17	9	9.5	9.5	8.5	5	39	1024	21	196	227
MP18	12	13.0	13.0	12.0	0	45	809	24	166	206
MP19	8	8.5	8.5	7.5	5	30	816	17	155	182
MP20	7	7.1	7.1	6.2	7	30	631	12	112	133
W30	15	13.9	13.9	13.0	1	72	113	1	1	2
W150	15	12.4	12.4	11.6	4	61	351	4	5	13
W666	16	15.1	15.1	14.2	6	79	1087	37	91	151
MB64	55	50.6	50.6	49.6	42	56	2551	8	30	43
MB1116	36	23.4	23.4	22.6	142	55	12823	255	4064	4391

Table 4: Strategy 1, partial pricing, $\mu_3 = 1$

Still using Strategy 1 for the computation of the trial step s_k, we have also tested the *partial pricing* routine that *completely* solves problem (2.12) (hence setting $\mu_3 = 1$). These results are reported in Table 4. The total cpu-times are better than those given in Table 3. This is due to much better cpu-times for the GCP calculations. Indeed, problems MP4 and MP18 for instance present similar numbers of GCP calculations and yet, the exact solution's calculation using partial pricing is less expensive than the approximate solution's calculation, even when $\mu_3 = 0.6$. This can be explained by the additional amount of work required for maintaining the upper bound on the value of $\mu_3 \alpha_k(t_i)$ in (2.18) when approximately solving (2.12). This additional work is not sufficiently balanced by the use of the upper bound and leads to the conclusion that it is not worth solving approximately the linear problem (2.12) in the GCP calculation, at least in the presence of *network* constraints, since a fast solver can then be implemented to solve problem (2.12) exactly. From now on, we therefore abandon the approximate solution of (2.12) in favour of the exact one using partial pricing.

We now analyse and compare the results obtained for the twenty-five test problems when using Strategy 1 or 2. These results are reported in Tables 4 and 5 respectively.

Problem	it	nf	ng	nH	np	gcp	cg	gcpcpu	cgcpu	totcpu
MP1	6	5.9	5.9	5.0	0	27	395	9	63	79
MP2	6	5.9	5.9	5.0	0	27	386	53	597	678
MP3	6	7.0	7.0	6.0	0	27	392	50	559	637
MP4	35	33.6	27.2	26.2	24	58	855	84	156	279
MP5	22	22.2	22.2	21.2	3	56	789	43	152	221
MP6	16	16.3	16.3	15.3	3	47	793	32	150	202
MP7	21	21.2	21.2	20.3	3	59	1460	44	275	344
MP8	21	20.9	20.9	20.0	2	78	4279	44	770	839
MP9	34	33.0	33.0	32.1	3	136	11160	70	2028	2139
MP10	7	7.5	7.5	6.6	2	25	258	3	12	18
MP11	16	16.3	16.3	15.3	3	47	793	32	149	201
MP12	29	28.7	25.9	24.9	18	97	2086	195	821	1091
MP13	8	8.3	8.3	7.3	2	27	516	13	30	50
MP14	8	5.3	5.3	4.8	1	30	417	9	30	52
MP15	15	15.3	15.3	14.4	3	40	654	30	123	172
MP16	16	16.3	16.3	15.3	3	47	793	32	150	202
MP17	16	16.3	16.3	15.3	3	47	793	32	150	202
MP18	12	13.0	13.0	12.0	0	45	818	25	163	203
MP19	16	16.3	16.3	15.3	3	47	793	32	149	201
MP20	23	22.1	22.1	21.2	5	66	806	38	141	206
W30	18	17.5	16.5	15.6	2	61	172	1	1	3
W150	14	11.7	11.7	10.9	4	55	344	4	4	11
W666	13	12.5	12.5	11.6	3	53	1008	29	63	112
MB64	18	18.1	18.1	17.1	3	25	634	3	6	10
MB1116	18	12.6	12.6	11.8	5	34	5621	134	1634	1802

Table 5: Strategy 2, partial pricing, $\mu_3 = 1$

The number of iterations used in the GCP calculations are generally quite reasonable when compared with the number of major iterations or with the total number of conjugate gradient recurrences, especially for Strategy 1. The same conclusion applies when comparing the respective cpu-times. This is partly due to the choice of a small value for λ in the GCP Algorithm. Indeed, we have tested the same code with $\lambda = 0.1$ replaced by $\lambda = 0.5$, and we have clearly detected a substantial increase in the number of iterations and the cpu-times for the GCP calculations. This thus justifies a choice for λ that allows a rapid branching to the RS Algorithm (Case 2 of Step 3 in the GCP Algorithm), therefore avoiding an unnecessarily high number of solutions of the linear network problem (2.12). Moreover, we also observe that the amount of work in the GCP calculations grows more slowly with the size of the problem than in the conjugate gradient scheme (compare MP10, MP11 and MP12 or the Dembo's problems, for instance). This is true for the number of iterations as well as for the cpu-times.

On the other hand, if we examine the behaviour of Strategies 1 and 2 on the twenty model problems, we observe in Table 5 a significant increase in the number of major iterations (and thus in the number of function, gradient and Hessian evaluations) that produces a significant increase in the number of GCP iterations (except for MP1, MP2, MP3 and MP18). This deterioration is due to the fact that Strategy 2 is not adapted to find the correct active set in an efficient way, resulting in more iterations to reach optimality. The loss produced is not balanced by the gain obtained in the number of pivoting phases and in the conjugate gradient recurrences (that increase slower and are much cheaper). We may then conclude that Strategy 2 is not adapted for handling problems that require an accurate strategy for the correct active set identification. However, we do not observe the same deterioration for problem MP18. This is due to the absence of bound constraints for this problem. We even see a slight improvement for problems MP2 and MP3. This can be explained by the fact that the correct active set at optimality is small (four arcs only) for these problems and is already contained in the active set at the starting point, and every subsequent iterate. Since no activation occurs during the optimization process, the two strategies are similar, while cheaper in terms of the amount of calculations in the conjugate gradient iteration for Strategy 2.

On Dembo's problems, the situation is reversed, except perhaps for W30 (which is too easy to tell), in that a clear amelioration now occurs when going from Strategy 1 to Strategy 2. Indeed, all iteration counts decrease, as do the cpu-times and the number of pivoting steps accordingly. Strategy 1 is less suitable to solve these problems because the active set at optimality is small compared to n and also because a lot of bounds are encountered during the conjugate gradient iteration (each time requiring a restart) and are thereafter immediately released. This phenomenon slows down convergence while requiring a substantial amount of extra work and may partly be explained by the fact that the strict complementarity slackness condition does not hold at the solution (except for MB64), so that the degenerate bounds may repeatedly enter and leave the active set from one conjugate gradient iteration to the next as the solution is approached.

We have also tested a third strategy that avoids pivoting steps during the conjugated gradient process by keeping feasibility of the iterates (with respect to the bound constraints (2.24)) *for the superbasic arcs only*. If used with restarting steps in the conjugate gradient iteration when a superbasic bound is encountered, this approach is essentially comparable to Strategy 1. On the other hand, we observed a clear deterioration when no restarting steps are allowed in the conjugate gradient iteration.

Considering the above comments, we may say that Strategy 1 outperforms Stratety 2 as soon as the problems to be solved require a careful handling of the constraints for the identification of the correct active set. Since the aim of this paper is to develop an efficient algorithm for the active set identification in the solution of large-scale nonlinear network problems, we therefore concentrate on the performances of Strategy 1 in the two next sections.

Finally, we have also tested the impact of the choice of the initial trust region radius Δ_0 on the performances of the method. Indeed, the initial value given in (3.3) is rather heuristic, and we actually observed that eighteen on the twenty-five test problems selected $\Delta_0 = 100$. Table 6 reports the results obtained when solving problem MB1116 for different initial trust region radii, with Strategy 1 or 2 and the GCP Algorithm with $\mu_3 = 1$ (economical version). These results, compared with those of Tables 4 and 5, show a possible saving of up to 25% in the total cpu-times, depending only on the value of Δ_0. This saving occurs essentially in the conjugate gradient iteration counts. This emphasizes the importance of a good choice for this last value.

Strategy	Δ_0	it	gcp	cg	gcpcpu	cgcpu	totcpu
ST1	10^5	35	55	10166	243	3033	3343
	10^9	34	56	10805	241	3373	3678
	10^{14}	34	57	10645	240	3238	3542
	10^{18}	34	57	10645	239	3244	3547
ST2	10^5	17	31	5144	127	1446	1605
	10^9	17	31	4614	126	1283	1440
	10^{14}	18	32	4683	132	1221	1386
	10^{18}	18	32	4683	133	1221	1386

Table 6: The effect of varying the initial trust region radius for MB1116

3.2 Variation of the test problems' features

We briefly interpret here our results on the twenty model test problems of Section 3.1 when varying the six items mentioned at the beginning of Section 3. The reader is invited to consult Table 4 in order to confirm the comments given below for Strategy 1.

We first observe essentially identical behaviour for the method of this manuscript and that of [22] when using the three different storage schemes for the Hessian matrix (see MP1 to MP3). Our cpu-times are clearly in favour of the internal storage technique, although, for example, the additional subroutine calls necessary in this context can be quite significant. We also observe a small increase in function, gradient and Hessian counts when going from the elemental dimension storage to that of one element, the number of conjugate gradient steps and the iteration counts being approximately unchanged. This effect is due to the loss of the partially separable character of the objective in the latter case, which prevents partial evaluations of the function or of its derivatives. The gains in cpu-time for the storage using one element as opposed to the elemental dimension storage is caused by the fact that the products involving the Hessian matrices are cheaper to compute (see [21]).

We also see, as in [21]–[22], that the method is sensitive to variations of conditioning (see MP4 to MP9). This is due to the use of the conjugate gradient method which is a conditioning sensitive method, and leads to an increase in the numbers of conjugate gradient recurrences (while the GCP calculations remain comparable).

The problem becomes slightly more difficult when its size increases, mostly because of the added complexity of the bound constraints (see for example MP10 to MP12). Nevertheless, the difficulty seems to increase faster in [21] than for our code. This will be confirmed in the next section.

Moreover, when the objective function is quadratic (i.e. when $a = 0$ in Table 2 or in (3.1)), we can say, unlike [21], that the problem is easier in terms of major iterations, function, gradient and Hessian evaluations, as well as in terms of conjugate gradient steps and cpu-times (see MP13 to compare with MP6, and MP14).

As observed in [21]–[22], a tighter requested accuracy on the solution does not cause a large increase in the number of major iterations (see MP15 to MP17). This is explained by the rapid rate of convergence achieved by both methods. The number of conjugate gradient recurrences may however be significantly increased by a tighter accuracy requirement, because a large part of this computational effort occurs in the last iterations of the algorithm, where the linear system (2.23) must be solved accurately.

Finally, unlike [21]–[22], we note that the introduction of bounds does not increase the number of major iterations, but even decreases it (see MP18 to MP20). This will be discussed in the next section. On the other hand, as in [21]–[22], the number of pivoting steps increases with the tightness of the bounds. This is due to the fact that the basic variables are increasingly constrained.

3.3 Comparison with the LSNNO routine

In this section, we compare the TRNNO routine (Strategy 1, partial pricing, Δ_0 given by (3.3)) with the LSNNO code of Tuyttens, both tested on the same machine.

We first consider the results of Table 4 in comparison with those produced by LSNNO in [21]–[22] when using Newton's method without preconditioning. We observe, in most cases, a decrease in the number of major iterations for TRNNO (especially for problems MP9 and MP12). This implies fewer function, gradient and Hessian evaluations. On the other hand, the number of conjugate gradient recurrences generally increases, mainly because Strategy 1 allows the restarting of the conjugate gradient scheme. So we may conclude that the TRNNO code requires fewer iterations than the LSNNO code, but that one iteration is more expensive for TRNNO, due to the GCP calculations and the restarting steps in the conjugate gradient iterations. We observe better cpu-times for TRNNO on the large model test problems (MP12 and MP14) and when the bounds on the variables become tighter (MP18 to MP20). In order to investigate this issue further, we have tested both codes on the model test problem for different sizes and type of bounds, with fixed parameters $a = 1$, $c = 10^2$, $\eta_3 = 10^{-5}$ for the final accuracy, and using a storage with internal dimension (I). We report the results in Tables 8 to 11 and in Figures 2 to 8. The various sizes specified by the parameter ℓ are given in Table 7. We have selected three types of bounds:

Case $i = 1$. We impose that $r \le [x]_j \le \infty$ for all index j such that mod(j,3) = 0, all other variables being unconstrained. r is successively equal to 0.15, 0.35, 0.55 and 0.75.

Case $i = 2$. We impose that $r \le [x]_j \le \infty$ for all index j such that mod(j,3) = 1, all other variables being unconstrained. r is successively equal to 0, 0.5, 1 and 2.5.

Case $i = 3$. We impose that $r \le [x]_j \le \infty$ for all index j whose corresponding arc is on horizontal lines or alternate vertical lines (beginning at the first) of the grid, all other variables being unconstrained. r is successively equal to 0, 0.1 and 0.2.

For the two first cases, one third of the variables are constrained while this ratio increases to three quarters for Case $i = 3$.

ℓ	11	12	13	14	15	16	19	22
n	1104	1300	1512	1740	1984	2244	3120	4140
n_n	576	676	784	900	1024	1156	1600	2116

Table 7: Number of arcs and nodes for a given ℓ

Firstly we comment on the results given in Table 8 and Figures 2 and 3 for Case $i = 1$. In Table 8 and the following ones, "%act" denotes the percentage of active bounds at the solution (computed by LSNNO).

r	ℓ	it		%act
		TRNNO	LSNNO	
0.15	16	13	19	9.6
	19	9	28	11.4
	22	17	34	13.4
0.35	16	15	18	23.3
	19	9	32	24.6
	22	10	66	25.1
0.55	16	9	25	26.2
	19	7	41	26.6
	22	7	42	27.0
0.75	16	5	24	27.4
	19	8	63	27.6
	22	7	41	28.0

Table 8: Comparison with LSNNO for Case $i = 1$

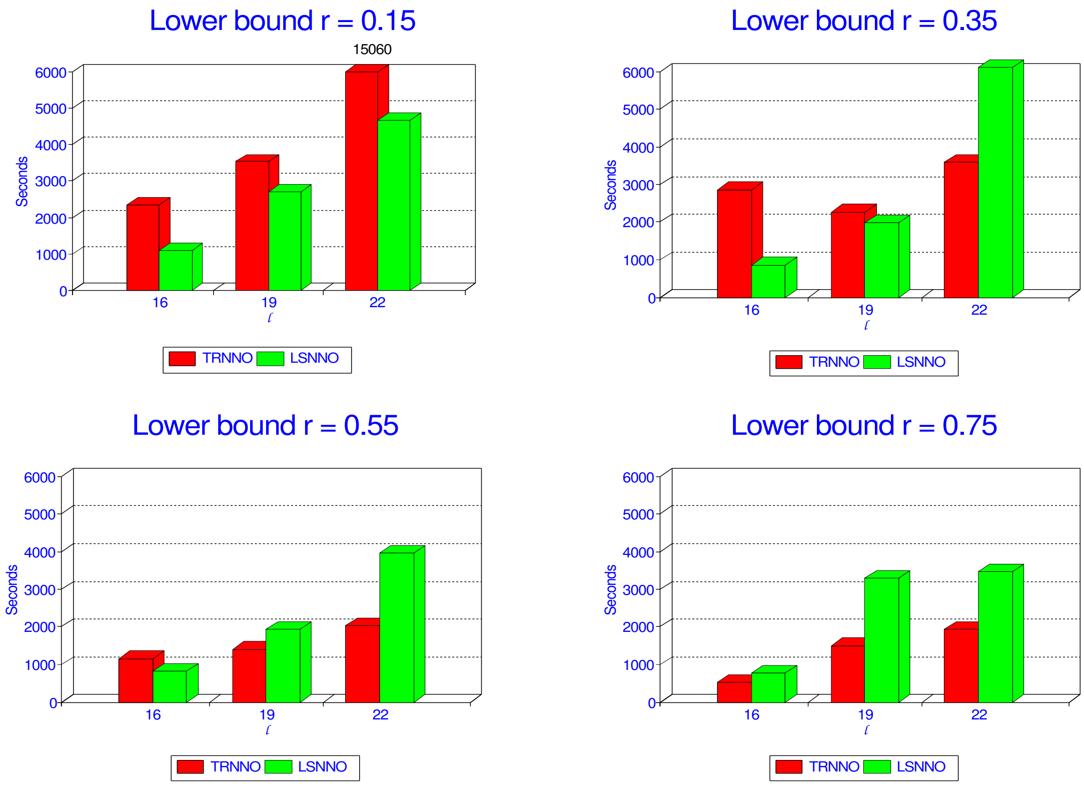

Figure 2: Comparison with LSNNO on cpu-times for fixed bounds (Case $i = 1$)

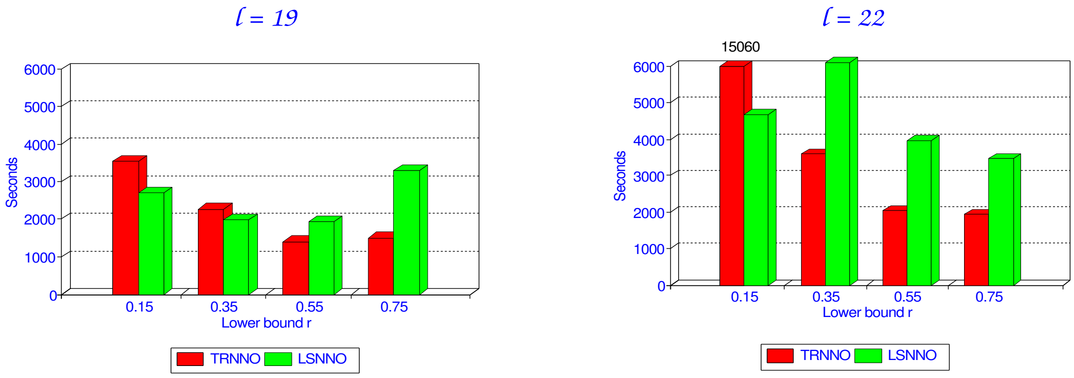

Figure 3: Comparison with LSNNO on cpu-times for fixed sizes (Case $i = 1$)

The results of Table 8 show that on the whole the number of major iterations decreases when bounds become tighter for TRNNO, as mentioned in the previous section. On the other hand, these numbers increase for LSNNO. Now comparing the cpu-times given in Figure 2, we may observe that this behaviour has the effect of improving the performances of TRNNO while those of LSNNO deteriorate. Moreover, we observe that TRNNO outperforms LSNNO for the largest problem first, then for the medium one and finally for the smallest one. Figure 3 once again confirms the above observation. For tighter and tighter bounds, TRNNO produces better and better cpu-times (except for $l = 19$ when going from $r = 0.55$ to $r = 0.75$), while those for LSNNO behave erratically but are consistently worse. Finally, from Figure 2 and the last column of Table 8, we can see that the cpu-times are overwhelmingly in favour of TRNNO when about a quarter of the bounds are active at optimality.

The second case is reported in Table 9 and Figures 4 and 5. These results corroborate the conclusions made for the previous case. It further shows (see Figure 4) how constant the number of iterations for TRNNO remains when the bounds and the size vary, while these numbers grow for LSNNO. Figure 5 displays this characteristic for $l = 22$.

r	ℓ	it		%act
		TRNNO	LSNNO	
	16	16	11	0.6
0.0	19	12	19	0.7
	22	21	29	0.7
	16	8	29	26.1
0.5	19	6	28	26.8
	22	8	39	27.2
	16	10	41	28.9
1.0	19	7	53	28.9
	22	7	63	29.3
	16	8	50	30.9
2.5	19	9	42	31.1
	22	10	103	31.3

Table 9: Comparison with LSNNO for Case $i = 2$

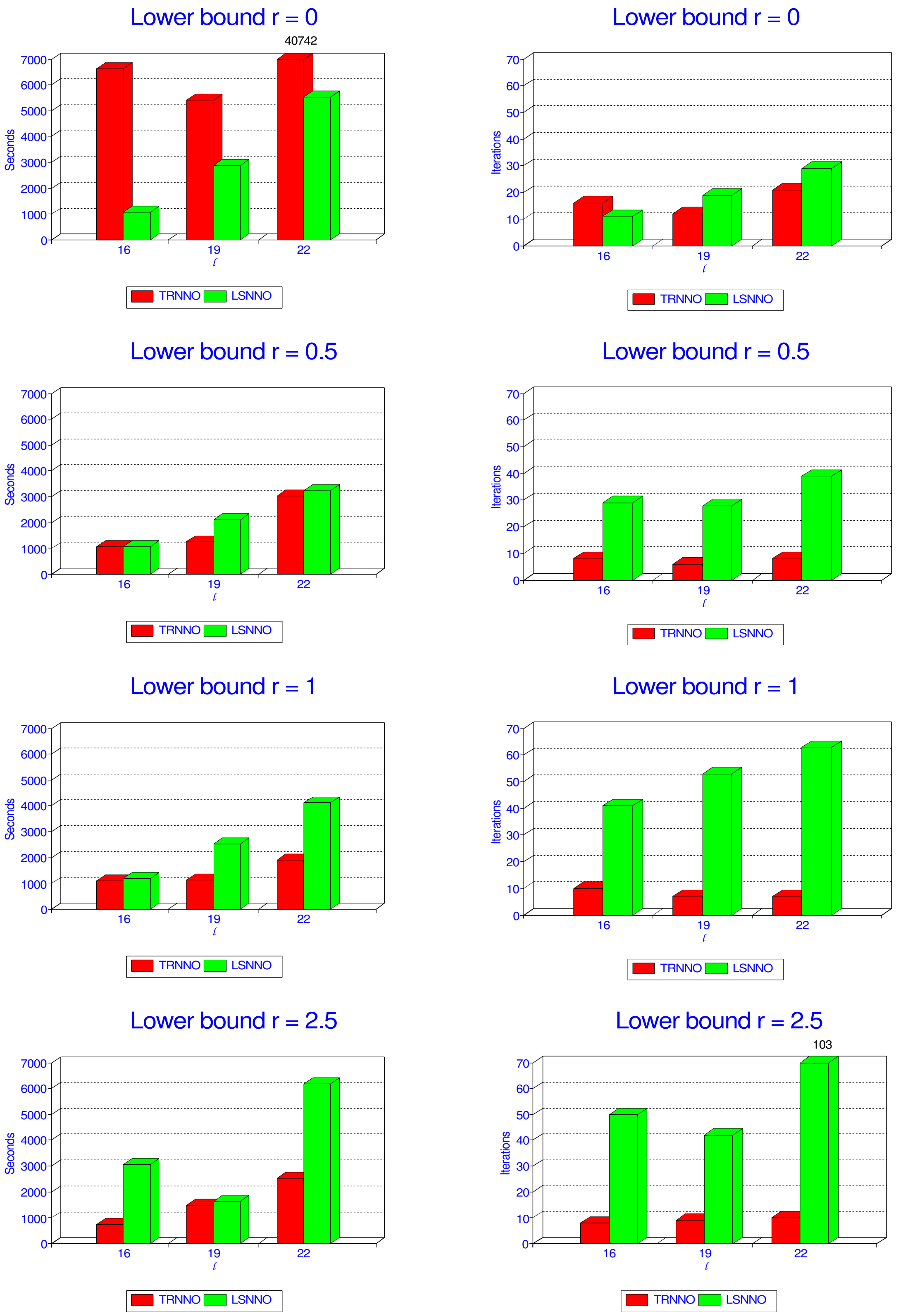

Figure 4: Comparison with LSNNO on cpu-times and iterations for fixed bounds (Case $i = 2$)

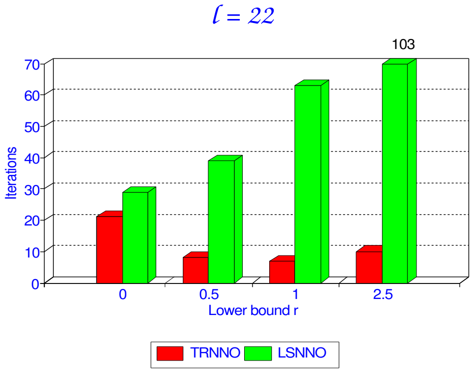

Figure 5: Comparison with LSNNO on iterations for fixed size (Case $i = 2$)

Finally, Table 10 and Figure 6 show the results for the third case of bounds. The absence of results for LSNNO means that it stopped with a flag error before having solved the problem. The results also confirm the above comments, except that this time TRNNO outperforms LSNNO immediately, even when about one per cent of the bounds are active at optimality. In particular, Figure 6 clearly shows the uniform behaviour of TRNNO. Indeed, for the three different bounds, the iterations numbers stay alike while the cpu-times grow slowly with the dimension of the problem. The tightness of the bounds does not seem to affect the performances of the code. On the other hand, it is not possible to attribute the same stability to LSNNO.

r	ℓ	it		%act
		TRNNO	LSNNO	
0.0	11	13	54	1.1
	12	13	58	1.1
	13	12	68	1.1
	14	13	53	1.1
	15	13	150	3.4
	16	14	76	1.3
0.1	11	10	54	13.1
	12	12	52	14.0
	13	9	48	14.7
	14	12	101	14.7
	15	9	85	17.9
	16	11	132	17.1
0.2	11	9	83	28.6
	12	23	142	30.8
	13	9	159	35.2
	14	11	237	40.1
	15	9		
	16	8		

Table 10: Comparison with LSNNO for Case $i = 3$

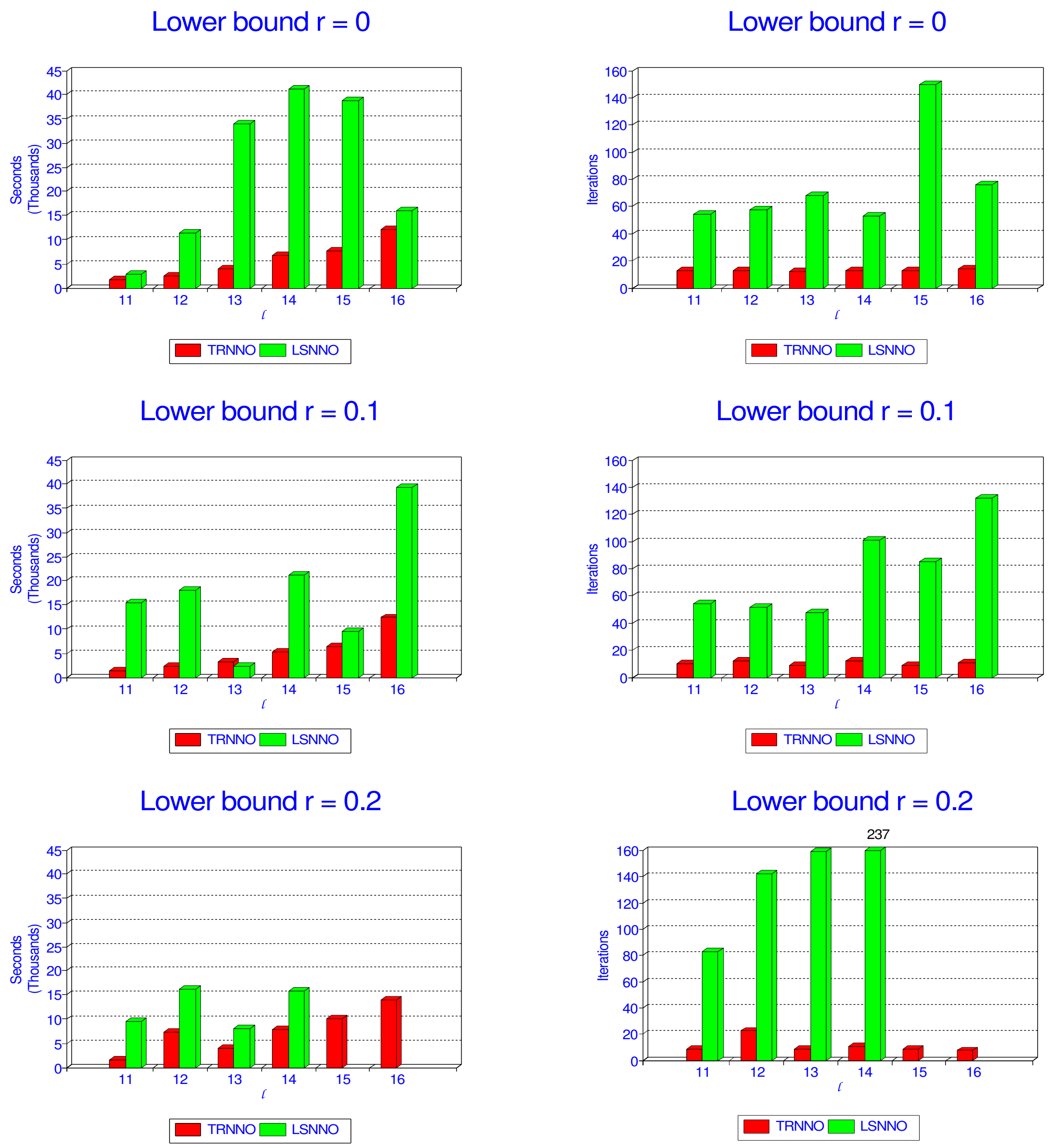

Figure 6: Comparison with LSNNO on cpu-times and iterations for fixed bounds (Case $i = 3$)

Moreover, although optimal function values usually agree for both codes in Cases $i = 1$ and 2, we have observed here a significant difference, always in the favor of TRNNO, for three quarters of the test problems. We also observed that the strict complementarity slackness condition did usually not hold at the solution for these problems.

Furthermore for Case $i = 3$, Table 11 and Figures 7 and 8 report the results for higher dimensions. They confirm the efficiency of TRNNO on large-scale problems. We also tested other cases of bounds which are not reported in this paper. They all corroborate the conclusions made in this section.

r	ℓ	it		%act
		TRNNO	LSNNO	
0.0	19	12	142	1.1
0.1	22	10	495	26.0

Table 11: Comparison with LSNNO for Case $i = 3$ (higher dimensions)

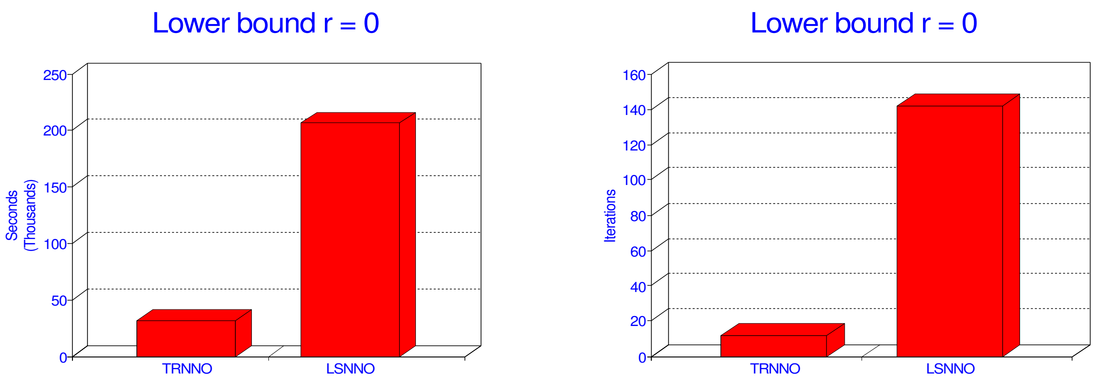

Figure 7: Comparison with LSNNO for $l = 19$ (Case $i = 3$)

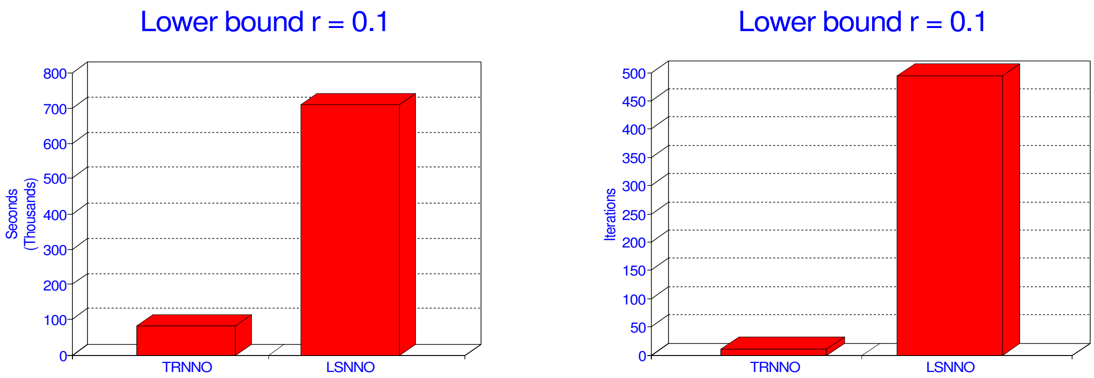

Figure 8: Comparison with LSNNO for $l = 22$ (Case $i = 3$)

4 Conclusions and perspectives

In this paper, we propose a new algorithm of trust region type to solve the nonlinear network problem (1.1). We consider practical implementation issues, including an explicit procedure for computing an approximate Generalized Cauchy Point and two strategies for calculating a candidate step at each iteration. Numerical tests are reported and discussed, showing the efficiency of the method, especially for large-scale problems and tight bound constraints. We believe that part of the success may be attributed to the ability of the GCP calculation to swiftly determine the set of (nondegenerate) active bounds at the solution.

The encouraging results show that the framework presented is worth considering for the solution of problem (1.1), especially in view of the good theoretical properties of the framework given in [6] and the numerical results for large problems. It also suggests some directions for future research and continued development. The method given here could be adapted for solving *general large-scale linearly constrained problems*. We could then envisage to produce effective methods for solving *general large-scale nonlinear programming problems* by combining the nonlinear constraints in a suitable fashion with the objective function (for instance in an augmented Lagrangian function [5], [7], [8]), and solving the resulting sequence of linearly constrained problems using the method described in this paper.

5 Acknowledgement

The author wish to thank Michel Bierlaire, Andy Conn, Nick Gould, Philippe Toint and Daniel Tuyttens for their useful comments and suggestions. Daniel Tuyttens also kindly gave access to his code, his test problems and his numerical results.

References

[1] D. P. Ahlfeld, R. S. Dembo, J. M. Mulvey, and S. A. Zenios. Nonlinear programming on generalized networks. *ACM Transactions on Mathematical Software*, 13(4):350–367, 1987.

[2] G. H. Bradley, G. G. Brown, and G. W. Graves. Design and implementation of large-scale transshipment algorithms. *Management Science*, 24:1–34, 1977.

[3] S. M. Carpenter, I. J. Lustig, J. M. Mulvey, and D. F. Shanno. Higher order predictor-corrector interior point methods with application to quadratic objectives. *SIAM Journal on Optimization*, (to appear), 1993.

[4] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. Global convergence of a class of trust region algorithms for optimization with simple bounds. *SIAM Journal on Numerical Analysis*, 25:433–460, 1988. See also same journal 26:764–767, 1989.

[5] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. A globally convergent augmented Lagrangian algorithm for optimization with general constraints and simple bounds. *SIAM Journal on Numerical Analysis*, 28(2):545–572, 1991.

[6] A. R. Conn, Nick Gould, A. Sartenaer, and Ph. L. Toint. Global convergence of a class of trust region algorithms for optimization using inexact projections on convex constraints. *SIAM Journal on Optimization*, 3(1):164–221, 1993.

[7] A. R. Conn, Nick Gould, A. Sartenaer, and Ph. L. Toint. Global convergence of two augmented Lagrangian algorithms for optimization with a combination of general equality and linear constraints. Technical Report 93/01, Department of Mathematics, FUNDP, Namur, Belgium, 1993.

[8] A. R. Conn, Nick Gould, A. Sartenaer, and Ph. L. Toint. Local convergence properties of two augmented Lagrangian algorithms for optimization with a combination of general equality and linear constraints. Technical Report 93/20, Department of Mathematics, FUNDP, Namur, Belgium, 1993.

[9] G. B. Dantzig. *Linear Programming and Extensions*. Princeton Uinversity Press, Princeton, USA, 1963.

[10] R. S. Dembo. The performance of NLPNET, a large scale nonlinear network optimizer. *Mathematical Programming, Series B*, 26:245–249, 1986.

[11] R. S. Dembo. A primal truncated Newton algorithm with application to large scale nonlinear network optimization. *Mathematical Programming, Series B*, 31:43–71, 1987.

[12] R. S. Dembo and J. G. Klincewicz. A scaled reduced gradient algorithm for network flow problems with convex separable costs. *Mathematical Programming*, 15:125–147, 1981.

[13] L. F. Escudero. A motivation for using the truncated Newton approach in a very large scale nonlinear network problem. *Mathematical Programming Studies*, 26:240–245, 1986.

[14] A. Griewank and Ph. L. Toint. On the unconstained optimization of partially separable functions. In M. J. D. Powell, editor, *Nonlinear Optimization 1981*, pages 301–312, London and New York, 1982. Academic Press.

[15] A. Griewank and Ph. L. Toint. Partitioned variable metric updates for large structured optimization problems. *Numerische Mathematik*, 39:119–137, 1982.

[16] M. D. Grigoriadis. An efficient implementation of the network simplex method. *Mathematical Programming*, 26:83–111, 1986.

[17] J. L. Kennington and R. V. Helgason. *Algorithms for Network Programming*. J. Wiley and Sons, New York, 1980.

[18] J. J. Moré. Recent developments in algorithms and software for trust region methods. In A. Bachem, M. Grötschel, and B. Korte, editors, *Mathematical Programming: The State of the Art*, pages 258–287, Berlin, 1983. Springer Verlag.

[19] B. A. Murtagh and M. A. Saunders. Large-scale linearly constrained optimization. *Mathematical Programming*, 14:41–72, 1978.

[20] Y. Sheffi. *Urban Transportation Networks*. Prentice-Hall, Englewood Cliffs, USA, 1985.

[21] Ph. L. Toint and D. Tuyttens. On large scale nonlinear network optimization. *Mathematical Programming, Series B*, 48(1):125–159, 1990.

[22] Ph. L. Toint and D. Tuyttens. LSNNO: a Fortran subroutine for solving large scale nonlinear network optimization problems. *ACM Transactions on Mathematical Software*, 18(3):308–328, 1992.

[23] D. Tuyttens and J. Teghem. Théorie des matroides et optimisation combinatoire. *Belgian Journal of Operations Research, Statistics and Computer Science*, 26(1):27–62, 1986.