Large-Scale Nonsmooth Optimization: New Variable Metric Bundle Algorithm with Limited Memory

M. Haarala K. Miettinen M. M. Mäkelä

Department of Mathematical Information Technology,
University of Jyväskylä, P.O. Box 35 (Agora),
FIN-40014 University of Jyväskylä, Finland.

Many practical optimization problems involve nonsmooth (that is, not necessarily differentiable) functions of hundreds or thousands of variables. In such problems, the direct application of smooth gradient-based methods may lead to a failure due to the nonsmooth nature of the problem. On the other hand, none of the current general nonsmooth optimization methods is efficient in large-scale settings. In this paper, we introduce a new limited memory variable metric -based bundle method for nonsmooth large-scale optimization. In addition, we introduce a new set of academic test problems for large-scale nonsmooth minimization. Finally, we give some encouraging results from numerical experiments using both academic and practical test problems.

Keywords: Nondifferentiable programming, large-scale optimization, bundle methods, variable metric methods, limited memory methods, test problems.

1 Introduction

In this paper, we describe a limited memory bundle algorithm for solving large nonsmooth (nondifferentiable) unconstrained optimization problems. We write this problem as

$$\begin{cases} \text{minimize} & f(\mathbf{x}) \\ \text{subject to} & \mathbf{x} \in \mathbb{R}^n, \end{cases} \tag{1}$$

where the objective function $f : \mathbb{R}^n \to \mathbb{R}$ is supposed to be locally Lipschitz continuous and the number of variables n is supposed to be large.

Nonsmooth optimization problems of type (1) arise in many fields of applications, for example, in image restoration (see, e.g., [10]) and in optimal control (see, e.g., [20]). The direct application of smooth gradient-based methods to nonsmooth problems may lead to a failure in convergence, in optimality conditions, or in gradient approximation (see, e.g., [11]). On the other hand, direct methods, for example, Powel's method (see, e.g., [3]) employing no derivative information, are quite unreliable and become inefficient when the size of the problem increases.

The methods for solving nonsmooth optimization problems can be divided into two main classes: subgradient methods (see, e.g., [22]) and bundle methods (see, e.g., [7, 8, 16, 19, 20]). They are all based on the assumption that at every point $\mathbf{x} \in \mathbb{R}^n$, we can evaluate the value of the objective function $f(\mathbf{x})$ and an arbitrary subgradient $\boldsymbol{\xi} \in \mathbb{R}^n$ from the subdifferential (see [2])

$$\partial f(\mathbf{x}) = \mathrm{conv}\{ \lim_{i \to \infty} \nabla f(\mathbf{x}_i) \mid \mathbf{x}_i \to \mathbf{x} \text{ and } \nabla f(\mathbf{x}_i) \text{ exists} \}, \tag{2}$$

where "conv" denotes the convex hull of a set.

At the moment, bundle methods are regarded as the most effective and reliable methods for nonsmooth optimization (see, e.g., [19]). Thus, we begin by giving a short review of standard bundle methods to point out the basic ideas and to motivate the need of solvers for large-scale nonsmooth optimization problems.

The basic idea of bundle methods is to approximate the subdifferential of the objective function by gathering the subgradients from previous iterations into a bundle. We suppose that, in addition to the current iteration point $\mathbf{x}_k$, we have some auxiliary points $\mathbf{y}_j \in \mathbb{R}^n$ (from previous iterations) and a bundle of subgradients $\boldsymbol{\xi}_j \in \partial f(\mathbf{y}_j)$ available for $j \in \mathcal{J}_k$, where $\mathcal{J}_k$ is a nonempty subset of $\{1, \ldots, k\}$. We approximate the objective function f by using a piecewise linear function

$$\hat{f}_k(\mathbf{x}) = \max_{j \in \mathcal{J}_k} \{ f(\mathbf{x}_k) + \boldsymbol{\xi}_j^T (\mathbf{x} - \mathbf{x}_k) - \beta_j^k \}, \tag{3}$$

where $\beta_j^k \geq 0$ is a so-called subgradient locality measure (see, e.g., [20]). A search direction can be determined as

$$\mathbf{d}_k = \arg\min_{\mathbf{d} \in \mathbb{R}^n} \{ \hat{f}_k(\mathbf{x}_k + \mathbf{d}) + \tfrac{1}{2} \mathbf{d}^T M_k \mathbf{d} \}, \tag{4}$$

where the role of the stabilizing term $\frac{1}{2}\mathbf{d}^T M_k \mathbf{d}$ is to guarantee the existence of the solution $\mathbf{d}_k$ and to keep the approximation local enough. The regular and symmetric $n \times n$ -matrix M_k is intended to accumulate information about

the curvature of the objective function f in a ball around $\mathbf{x}_k$. In the most frequently used proximal bundle method, the matrix M_k is diagonal of the form $M_k = u_k I$, where I is the identity matrix and the weighting parameter $u_k > 0$ (see, e.g., [20]).

The minimization of problem (4) is equivalent (see, e.g., [20]) to the (smooth) quadratic subproblem of finding the solution $(\mathbf{d}_k, v_k) \in \mathbb{R}^{n+1}$ of

$$\begin{cases} \text{minimize} & \frac{1}{2}\mathbf{d}^T M_k \mathbf{d} + v \\ \text{subject to} & -\beta_j^k + \mathbf{d}^T \boldsymbol{\xi}_j \leq v \quad \text{for all } j \in \mathcal{J}_k. \end{cases} \tag{5}$$

By duality this is equivalent to finding Lagrange multipliers λ_j^k for $j \in \mathcal{J}_k$ that solve the quadratic dual problem

$$\begin{cases} \text{minimize} & \frac{1}{2}\left(\sum\limits_{j\in\mathcal{J}_k} \lambda_j \boldsymbol{\xi}_j\right)^T M_k^{-1} \left(\sum\limits_{j\in\mathcal{J}_k} \lambda_j \boldsymbol{\xi}_j\right) + \sum\limits_{j\in\mathcal{J}_k} \lambda_j \beta_j^k \\ \text{subject to} & \sum\limits_{j\in\mathcal{J}_k} \lambda_j = 1 \quad \text{and} \\ & \lambda_j \geq 0, \quad \text{for all } j \in \mathcal{J}_k. \end{cases} \tag{6}$$

A new auxiliary point $\mathbf{y}_{k+1}$ is defined by $\mathbf{y}_{k+1} = \mathbf{x}_k + t_R^k \mathbf{d}_k$, where $t_R^k \in (0, 1]$ is an appropriately chosen step size (see, e.g., [8]). A serious step

$$\mathbf{x}_{k+1} = \mathbf{y}_{k+1} \tag{7}$$

is taken if $\mathbf{y}_{k+1}$ is significantly better than $\mathbf{x}_k$, in other words,

$$f(\mathbf{y}_{k+1}) \leq f(\mathbf{x}_k) + \varepsilon_L t_R^k v_k, \tag{8}$$

where $\varepsilon_L \in (0, 1/2)$ is a fixed line search parameter and v_k, which is the solution of the problem (5), represent the predicted descent of f at $\mathbf{x}_k$ (see, e.g., [19]). Otherwise, a null step is taken that keeps the current iteration point unchanged ($\mathbf{x}_{k+1} = \mathbf{x}_k$) but the information about the objective function is increased by adding a new subgradient $\boldsymbol{\xi}_{k+1} \in \partial f(\mathbf{y}_{k+1})$ into a bundle. The global convergence of bundle methods with a limited number of stored subgradients can be guaranteed by using a subgradient aggregation strategy, which accumulates information from the previous iterations (see [8]).

In their present form, bundle methods are efficient for small- and medium-scale problems. However, their computational demand expands in large-scale problems with more than 1000 variables. This is explained by the fact that bundle methods need relatively large bundles to be capable of solving the problems efficiently. In other words, the size of the bundle has to be

approximately the same as the number of variables and, thus, the quadratic subproblem (6) becomes very time-consuming to solve.

In variable metric bundle methods introduced by Lukšan and Vlček [14, 24], the search direction is calculated by using the variable metric approximation of the inverse of the Hessian matrix ($D_k = M_k^{-1}$ in (6)). The idea of the methods is to use only three subgradients: one calculated at the current iteration point $\mathbf{x}_k$, one calculated at the new auxiliary point $\mathbf{y}_{k+1}$, and an aggregated one, containing information from previous iterations. Thus, the dimension of the normally time-consuming quadratic subproblem (6) is only three and it can be solved with simple calculations. However, variable metric bundle methods use dense approximations of the Hessian matrix to calculate the search direction and, thus, also these methods become inefficient when the dimension of the problem increases.

We have not found any general bundle-based solver for large-scale nonsmooth optimization problems from the literature. Thus, we can say that at the moment the only possibility to optimize nonsmooth large-scale problems is to use some subgradient methods. However, the basic subgradient methods suffer from some serious disadvantages: a nondescent search direction may occur, there exists no implementable stopping criterion, and the convergence speed is poor (not even linear) (see, e.g., [11]). On the other hand, the more advanced variable metric based subgradient methods (see, e.g., [22]) using dense matrices suffer from the same drawbacks than the variable metric bundle methods. This means that there is an evident need of reliable and efficient solvers for nonsmooth large-scale optimization problems.

In this paper, we introduce a new limited memory bundle method for large-scale nonsmooth unconstrained minimization. The method is a hybrid of the variable metric bundle methods [14, 24] and the limited memory variable metric methods (see, e.g., [1, 21]), where the latter have been developed for smooth large-scale optimization. The new method uses all the ideas of the variable metric bundle method but the search direction is calculated using a limited memory approach. Thus, the time-consuming quadratic subproblem (6) appearing in the standard bundle methods need not to be solved. Furthermore, we use only few vectors to represent the variable metric approximation of the Hessian matrix and, thus, we avoid storing and manipulating large matrices as is the case in variable metric bundle methods. These improvements make the limited memory bundle method suitable for large-scale optimization. Namely, the number of operations needed for the calculation of the search direction and the aggregate values is only linearly dependent on the number of variables while, for example, with the original variable metric bundle method, this dependence is quadratic.

This paper is organized as follows: In the following section we introduce the new limited memory bundle method. In Section 3, we introduce a new set of academic test problems for large-scale nonsmooth minimization. Then, in Section 4, we analyze numerical experiments concerning some existing bundle methods and our new method. The numerical results to be presented demonstrate the usability of the new method with both smooth and nonsmooth large-scale minimization problems. Finally, in Section 5, we conclude by giving a short summary of the performance of the methods tested.

2 Limited Memory Bundle Method

In this section, we present a new method for large-scale nonsmooth unconstrained optimization. The method will be called the limited memory bundle method and its basic idea is very simple. We use all the ideas of variable metric bundle methods but the variable metric approximation of the Hessian matrix is calculated by using a limited memory approach. This means that we use some ideas essential to bundle methods, namely the utilization of null steps, simple aggregation of subgradients, and subgradient locality measures, but the search direction is calculated by using the limited memory variable metric updates. The basic idea of this limited memory approach is that the variable metric update of the approximated Hessian is not constructed explicitly. The updates use the information of the last few iterations to implicitly define a variable metric approximation. In practice, this means that the approximation of the Hessian matrix is not as accurate as that of the original variable metric bundle methods but both the storage space required and the number of operations used are significantly smaller. In smooth large-scale settings, there exist also some other possibilities to deal with the variable metric approximation of the Hessian matrix (see, e.g., [4, 23]). However, we chose this limited memory approach to be adopted for nonsmooth problems because it does not need any information of the structure of the problem or its Hessian. Thus, the only assumption required is that the objective function f is locally Lipschitz continuous.

Next, we go through the algorithm step by step and describe both its theoretical properties and some details of the implementation. In what follows, we use the following notations: the current iteration is denoted by k and the approximation of the inverse of the Hessian matrix is denoted by D, the subgradient of the objective function is denoted by $\boldsymbol{\xi}$ and an aggregate subgradient to be described in Subsection 2.3 is denoted by $\tilde{\boldsymbol{\xi}}$.

2.1 Direction Finding

In this subsection, we describe how to find the search direction $\mathbf{d}_k$ by using the limited memory bundle method. The basic idea in direction finding is the same as with the limited memory variable metric methods (see, e.g., [1]) and the approximations D_k are formed implicitly by using the limited memory variable metric updates.

However, due to the usage of null steps some modifications similar to variable metric bundle methods [14, 24] have to be made: After a null step, the approximation D_k is formed by using the limited memory SR1 update (see, e.g., [1]), since this update formula gives us a possibility of preserving the boundedness of the generated matrices (that is, the eigenvalues of the matrix lie in the compact interval not containing zero) (see, e.g., [24]). In addition, we use an aggregate subgradient $\tilde{\boldsymbol{\xi}}_k$ to calculate the search direction

$$\mathbf{d}_k = -D_k \tilde{\boldsymbol{\xi}}_k. \tag{9}$$

Because the boundedness of the generated matrices is not required after a serious step (see [24]), the more efficient limited memory BFGS update formula (see, e.g., [1]) is used to compute the approximation D_k of the inverse of the Hessian. The search direction $\mathbf{d}_k$ is calculated by using the original subgradient $\boldsymbol{\xi}_k \in \partial f(\mathbf{x}_k)$. Thus, after a serious step, the search direction is defined by

$$\mathbf{d}_k = -D_k \boldsymbol{\xi}_k. \tag{10}$$

Note that the matrix D_k is not formed explicitly but the search direction $\mathbf{d}_k$ is calculated using the limited memory approach to be described in Subsection 2.6.

2.2 Line Search

Next, we consider how to calculate a new iteration point $\mathbf{x}_{k+1}$ when the search direction $\mathbf{d}_k$ has been calculated. Similarly to the standard bundle methods and the variable metric bundle methods, we use the line search procedure (see, e.g., [24]) that generates two points

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{x}_k + t_L^k \mathbf{d}_k \qquad \text{and} \\ \mathbf{y}_{k+1} &= \mathbf{x}_k + t_R^k \mathbf{d}_k, \end{aligned} \tag{11}$$

where $t_R^k \in (0, t_I^k]$, $t_L^k \in [0, t_R^k]$ are step sizes, $t_I^k \in [t_{min}, t_{max})$ is the initial step size, and $t_{max} > 1$ and $t_{min} \in (0, 1)$ are the upper and the lower bounds

for the initial step size t_I^k, respectively. Note that in standard bundle methods the initial step size t_I^k is equal to 1 at every iteration (see, e.g., [20]). However, similarly to the original variable metric bundle method, we have the possibility to use step sizes greater than 1 here, since the information about the objective function included in the matrix D_k, is not sufficient for a proper step size determination in the nonsmooth case (see [24]). The initial step size t_I^k is selected by using a bundle containing auxiliary points and corresponding function values and subgradients. The procedure used is exactly the same as in the original variable metric bundle method for nonconvex objective functions. The detailed description of the selection of this step size can be found in [24].

A necessary condition for a serious step to be taken is to have

$$t_R^k = t_L^k > 0 \qquad \text{and} \qquad f(\mathbf{y}_{k+1}) \leq f(\mathbf{x}_k) - \varepsilon_L t_R^k w_k, \tag{12}$$

where $\varepsilon_L \in (0, 1/2)$ is a fixed line search parameter and $w_k > 0$ represents the desirable amount of descent of f at $\mathbf{x}_k$ (note that this parameter is not the same as v_k in (8)). If the required condition (12) is satisfied, we set

$$\mathbf{x}_{k+1} = \mathbf{y}_{k+1} \tag{13}$$

and a serious step is taken. A null step is taken, if

$$t_R^k > t_L^k = 0 \qquad \text{and} \qquad -\beta_{k+1} + \mathbf{d}_k^T \boldsymbol{\xi}_{k+1} \geq -\varepsilon_R w_k, \tag{14}$$

where $\varepsilon_R \in (\varepsilon_L, 1)$ is a fixed line search parameter, $\boldsymbol{\xi}_{k+1} \in \partial f(\mathbf{y}_{k+1})$, and β_{k+1} is the subgradient locality measure similar to bundle methods (see, e.g., [19]), that is,

$$\beta_{k+1} = \max\{|f(\mathbf{x}_k) - f(\mathbf{y}_{k+1}) + (\mathbf{y}_{k+1} - \mathbf{x}_k)^T \boldsymbol{\xi}_{k+1})|,\, \gamma \|\mathbf{y}_{k+1} - \mathbf{x}_k\|^\omega \}, \tag{15}$$

where $\gamma \geq 0$ is a distance measure parameter and $\omega \geq 1$ is a locality measure parameter supplied by the user. The distance measure parameter γ can be set to zero when f is convex.

In the case of a null step, we set

$$\mathbf{x}_{k+1} = \mathbf{x}_k \tag{16}$$

but information about the objective function is increased because of the auxiliary point $\mathbf{y}_{k+1}$ and the auxiliary subgradient $\boldsymbol{\xi}_{k+1} \in \partial f(\mathbf{y}_{k+1})$ that we store.

The step sizes t_L^k and t_R^k for the limited memory bundle method can be determined by using the line search procedure quite similar to that at the

original variable metric bundle method [24]. However, in order to avoid many consecutive null steps, we have added an additional interpolation step. That is, we look for more suitable step sizes t_L^k and t_R^k by using an extra interpolation loop if necessary. The role of this additional step is that if we have already taken a null step at the previous iteration, we rather try to find a step size suitable for a serious step (that is, so that (12) is valid) even if the condition (14) required for a null step was satisfied.

Note that under some semismoothness assumptions, the line search procedure used is guaranteed to find the step sizes t_L^k and t_R^k such that exactly one of the two possibilities, serious step or null step, occurs (see [24]).

2.3 Subgradient Aggregation

In this subsection, we describe the aggregation procedure used with the limited memory bundle method. In principle, the aggregation procedure is the same as that with the original variable metric bundle methods [14, 24]. However, since the matrix D_k is not formed explicitly here, the practical implementation of the aggregation procedure differs from that of the original method.

The aggregation procedure uses three subgradients and two locality measures. We denote by m the lowest index j satisfying $\mathbf{x}_j = \mathbf{x}_k$ (that is, m is the index of the iteration after the latest serious step). Suppose that we have the current subgradient $\boldsymbol{\xi}_m \in \partial f(\mathbf{x}_k)$, the auxiliary subgradient $\boldsymbol{\xi}_{k+1} \in \partial f(\mathbf{y}_{k+1})$, and the current aggregate subgradient $\tilde{\boldsymbol{\xi}}_k$ (note that $\tilde{\boldsymbol{\xi}}_1 = \boldsymbol{\xi}_1$) available. In addition, suppose that we have the current locality measure β_{k+1} (see (15)) and the current aggregate locality measure $\tilde{\beta}_k$ from the previous iteration (note that $\tilde{\beta}_1 = 0$). The quite complicated quadratic subproblem (6) appearing in the standard bundle methods reduces to the minimization of the function

$$\begin{aligned}\varphi(\lambda_1, \lambda_2, \lambda_3) = (\lambda_1\boldsymbol{\xi}_m + \lambda_2\boldsymbol{\xi}_{k+1} + \lambda_3\tilde{\boldsymbol{\xi}}_k)^T D_k(\lambda_1\boldsymbol{\xi}_m + \lambda_2\boldsymbol{\xi}_{k+1} + \lambda_3\tilde{\boldsymbol{\xi}}_k) \\ + 2(\lambda_2\beta_{k+1} + \lambda_3\tilde{\beta}_k),\end{aligned} \tag{17}$$

where $\lambda_i \geq 0$ for $i \in \{1, 2, 3\}$ and $\sum_{i=1}^3 \lambda_i = 1$. The optimal values λ_i^k, $i \in \{1, 2, 3\}$ can be calculated by using simple formulae (see [24]). However, since we do not form the matrix D_k explicitly, we have to use limited memory BFGS (the first null step after any serious step) or SR1 (more than one consecutive null steps) update to implicitly define the approximation of the inverse of the Hessian matrix.

Now the next aggregate subgradient $\tilde{\boldsymbol{\xi}}_{k+1}$ is defined as a convex combination of the three subgradients mentioned above:

$$\tilde{\boldsymbol{\xi}}_{k+1} = \lambda_1^k \boldsymbol{\xi}_m + \lambda_2^k \boldsymbol{\xi}_{k+1} + \lambda_3^k \tilde{\boldsymbol{\xi}}_k \tag{18}$$

and the next aggregate locality measure $\tilde{\beta}_{k+1}$ as a convex combination of the two locality measures:

$$\tilde{\beta}_{k+1} = \lambda_2^k \beta_{k+1} + \lambda_3^k \tilde{\beta}_k. \tag{19}$$

Note that the aggregate values are computed only if the last step was a null step. Otherwise, we set $\tilde{\boldsymbol{\xi}}_{k+1} = \boldsymbol{\xi}_{k+1} \in \partial f(\mathbf{x}_{k+1})$ and $\tilde{\beta}_{k+1} = 0$.

2.4 Stopping Criterion

For smooth functions, a necessary condition for a local minimum is that the gradient has to be zero and by continuity it becomes small when we are close to an optimal point. This is no longer true when we replace the gradient by an arbitrary subgradient. Due to the subgradient aggregation, we have quite a useful approximation to the gradient, namely the aggregate subgradient $\tilde{\boldsymbol{\xi}}_k$. However, as a stopping criterion, the direct test $\|\tilde{\boldsymbol{\xi}}_k\| < \varepsilon$, for some $\varepsilon > 0$, is too uncertain, if the current piecewise linear approximation (see (3)) is too rough. Therefore, we use the approximation D_k of the inverse of the Hessian matrix and the aggregate subgradient locality measure $\tilde{\beta}_k$ to improve the accuracy of the norm of the aggregate subgradient. The aggregate subgradient locality measure $\tilde{\beta}_k$ approximates the accuracy of the current linearization: If the value of the locality measure is large, then the linearization is rough. On the other hand, if the value is near zero, then the linearization is quite accurate and, thus, we can stop the algorithm if the norm of the aggregate subgradient is small enough.

Since in practice the matrix D_k is not formed explicitly we use the direction vector $\mathbf{d}_k = -D_k \tilde{\boldsymbol{\xi}}_k$ instead. Hence, the stopping parameter w_k at iteration k is defined by

$$w_k = -2\tilde{\boldsymbol{\xi}}_k^T \mathbf{d}_k + 4\tilde{\beta}_k. \tag{20}$$

The multipliers 2 and 4 in (20) are chosen experimentally such that the accuracy of the new method would be approximately the same as with the other bundle methods. Note that the parameter w_k is also used during the line search procedure (see (12)) to represent the desirable amount of descent.

This first part of our stopping criterion is quite similar to that of the original variable metric bundle method. However, in practice the limited memory approximation D_k of the inverse of the Hessian matrix is not very accurate and computational experiments showed that some accidental terminations may occur (that is, the optimization was terminated before the minimum point was actually achieved). Thus, we added a second stopping parameter q_k, which does not depend on the matrix D_k. The second stopping parameter is similar to that in standard bundle methods (see, e.g. [20]), that is

$$q_k = \frac{1}{2}\tilde{\boldsymbol{\xi}}_k^T \tilde{\boldsymbol{\xi}}_k + \tilde{\beta}_k. \tag{21}$$

Now, the stopping criterion is given by:

$$\text{If } w_k < \varepsilon \text{ and } q_k < 10^3\varepsilon, \text{ for given } \varepsilon > 0, \text{ then stop.} \tag{22}$$

As before, the multiplier 10^3 above is chosen experimentally.

2.5 Algorithm

We are now ready to present the limited memory bundle method for nonsmooth large-scale unconstrained optimization.

Algorithm 1. (Limited Memory Bundle Method).

Data: Select the upper and the lower bounds $t_{max} > 1$ and $t_{min} \in (0, 1)$ for serious steps. Select positive line search parameters $\varepsilon_L \in (0, 1/2)$ and $\varepsilon_R \in (\varepsilon_L, 1)$. Choose the final accuracy tolerance $\varepsilon > 0$, the distance measure parameter $\gamma \geq 0$ ($\gamma = 0$ if f is convex), and the locality measure parameter $\omega \geq 1$.

Step 0: (*Initialization.*) Choose a starting point $\mathbf{x}_1 \in \mathbb{R}^n$. Set $\beta_1 = 0$ and $\mathbf{y}_1 = \mathbf{x}_1$. Compute

$$f_1 = f(\mathbf{x}_1) \qquad \text{and}$$
$$\boldsymbol{\xi}_1 \in \partial f(\mathbf{x}_1).$$

Set the iteration counter $k = 1$.

Step 1: (*Serious step initialization.*) Set the aggregate subgradient $\tilde{\boldsymbol{\xi}}_k = \boldsymbol{\xi}_k$ and the aggregate subgradient locality measure $\tilde{\beta}_k = 0$. Set an index for the serious step $m = k$.

Step 2: (*Direction finding.*) Compute

$$\mathbf{d}_k = -D_k \tilde{\boldsymbol{\xi}}_k$$

by a limited memory BFGS update, if $m = k$ (Algorithm 3) and by a limited memory SR1 update, otherwise (Algorithm 2). Note that $\mathbf{d}_1 = -\tilde{\boldsymbol{\xi}}_1$.

Step 3: (*Stopping criterion.*) Calculate w_k and q_k by (20) and (21), respectively. If $w_k < \varepsilon$ and $q_k < 10^3\varepsilon$, then stop.

Step 4: (*Line search.*) Calculate the initial step size $t_I^k \in [t_{min}, t_{max})$. Determine the step sizes $t_R^k \in (0, t_I^k]$ and $t_L^k \in [0, t_R^k]$ to take either a serious step or a null step (that is, check whether (12) or (14) is valid). Set the corresponding values

$$\begin{aligned}
\mathbf{x}_{k+1} &= \mathbf{x}_k + t_L^k \mathbf{d}_k, \\
\mathbf{y}_{k+1} &= \mathbf{x}_k + t_R^k \mathbf{d}_k, \\
f_{k+1} &= f(\mathbf{x}_{k+1}), \\
\boldsymbol{\xi}_{k+1} &\in \partial f(\mathbf{y}_{k+1}).
\end{aligned}$$

Set $\mathbf{u}_k = \boldsymbol{\xi}_{k+1} - \boldsymbol{\xi}_m$ and $\mathbf{s}_k = \mathbf{y}_{k+1} - \mathbf{x}_k = t_R^k \mathbf{d}_k$. If $t_L^k > 0$ (serious step), set $\beta_{k+1} = 0$, $k = k+1$, and go to Step 1. Otherwise, calculate the locality measure β_{k+1} by (15).

Step 5: (*Aggregation.*) Determine multipliers $\lambda_i^k \geq 0$, $i \in \{1, 2, 3\}$, $\sum_{i=1}^3 \lambda_i^k = 1$ that minimize the function (17), where D_k is obtained by the limited memory BFGS update, if $m = k$ and by the limited memory SR1 update, otherwise. Set

$$\begin{aligned}
\tilde{\boldsymbol{\xi}}_{k+1} &= \lambda_1^k \boldsymbol{\xi}_m + \lambda_2^k \boldsymbol{\xi}_{k+1} + \lambda_3^k \tilde{\boldsymbol{\xi}}_k \qquad \text{and} \\
\tilde{\beta}_{k+1} &= \lambda_2^k \beta_{k+1} + \lambda_3^k \tilde{\beta}_k.
\end{aligned}$$

Set $k = k + 1$ and go to Step 2.

Note that in Steps 2 and 5, the matrices D_k are not formed explicitly but the search direction $\mathbf{d}_k$, and the aggregate values $\tilde{\boldsymbol{\xi}}_{k+1}$, and $\tilde{\beta}_{k+1}$ are calculated using the difference vectors $\mathbf{u}_k$ and $\mathbf{s}_k$.

As mentioned in Subsection 2.3, the aggregation procedure uses only three subgradients and two locality measures to calculate the new aggregate values. In practice, this means that the minimum size of the bundle m_ξ is 2 and a larger bundle is used only for the selection of the initial step size (see [24]).

2.6 Matrix Updating

Finally, we need to consider how to update the approximation D_k of the inverse of the Hessian matrix and thus, how to find the search direction $\mathbf{d}_k$. Note that until this point, the procedures described are very similar to those of the original variable metric bundle method [24].

We use a compact representation of limited memory matrices (see [1]), since in addition to the BFGS updating formula, we need the SR1 updating formula and for SR1 updates, there exists no recursive updating formula analogous to that given in [21]. Moreover, the compact representation of limited memory matrices facilitates the possibility to generalize the method for constrained optimization.

The basic idea of the limited memory matrix updating is that instead of storing the matrices D_k, we use the information of the last few iterations to implicitly define the approximation of the inverse of the Hessian. This is done by storing a certain number of correction pairs $(\mathbf{s}_i, \mathbf{u}_i)$, $(i < k)$, where

$$\begin{aligned} \mathbf{s}_k &= \mathbf{y}_{k+1} - \mathbf{x}_k \qquad \text{and} \\ \mathbf{u}_k &= \boldsymbol{\xi}_{k+1} - \boldsymbol{\xi}_m. \end{aligned} \tag{23}$$

Here, as before, $\mathbf{y}_{k+1}$ is the current auxiliary iteration point, $\mathbf{x}_k$ is the current iteration point, and $\boldsymbol{\xi}_{k+1}$ and $\boldsymbol{\xi}_m$ are the corresponding subgradients of these points (m is the index of the iteration after the latest serious step). When the storage space available is used up, the oldest corrections are deleted to make room for new ones. All the subsequent iterations are of this form: one correction pair is deleted and a new one is inserted.

Let us denote by m_c the maximum number of stored corrections supplied by the user $(3 \leq m_c)$ and by $m_k = \min\{\, k-1, m_c \,\}$ the current number of stored corrections. We assume that the maximum number of stored corrections m_c is constant, although it is possible to adapt all the formulae of this subsection to the case where m_c varies at every iteration (see, e.g., [9]).

The $n \times m_k$ -dimensional correction matrices S_k and U_k are defined by

$$\begin{aligned} S_k &= \begin{bmatrix} \mathbf{s}_{k-m_k} & \dots & \mathbf{s}_{k-1} \end{bmatrix} \qquad \text{and} \\ U_k &= \begin{bmatrix} \mathbf{u}_{k-m_k} & \dots & \mathbf{u}_{k-1} \end{bmatrix}. \end{aligned} \tag{24}$$

These correction matrices are used to implicitly define the approximation of the inverse of the Hessian matrix at each iteration. When a new auxiliary iteration point $\mathbf{y}_{k+1}$ is generated, the new correction matrices S_{k+1} and U_{k+1} are obtained by deleting the oldest corrections $\mathbf{s}_{k-m_k}$ and $\mathbf{u}_{k-m_k}$ from S_k and

U_k if $m_{k+1} = m_k$ (that is, $k > m_c$) and by adding the most recent corrections $\mathbf{s}_k$ and $\mathbf{u}_k$ to the matrices. Thus, except for the first few iterations, we always have the m_c most recent correction pairs $(\mathbf{s}_i, \mathbf{u}_i)$ available.

We define the inverse limited memory BFGS update by the formula (see [1])

$$D_k = \vartheta_k I + \begin{bmatrix} S_k & \vartheta_k U_k \end{bmatrix} \begin{bmatrix} (R_k^{-1})^T (C_k + \vartheta_k U_k^T U_k) R_k^{-1} & -(R_k^{-1})^T \\ -R_k^{-1} & 0 \end{bmatrix} \begin{bmatrix} S_k^T \\ \vartheta_k U_k^T \end{bmatrix}. \tag{25}$$

Here, R_k is an upper triangular matrix of order m_k given by the form

$$(R_k)_{ij} = \begin{cases} (\mathbf{s}_{k-m_k-1+i})^T (\mathbf{u}_{k-m_k-1+j}) & \text{if } i \le j \\ 0 & \text{otherwise,} \end{cases} \tag{26}$$

C_k is a diagonal matrix of order m_k such that

$$C_k = \operatorname{diag}[\mathbf{s}_{k-m_k}^T \mathbf{u}_{k-m_k}, \ldots, \mathbf{s}_{k-1}^T \mathbf{u}_{k-1}], \tag{27}$$

and the multiplier $\vartheta_k > 0$ is given by

$$\vartheta_k = \frac{\mathbf{u}_{k-1}^T \mathbf{s}_{k-1}}{\mathbf{u}_{k-1}^T \mathbf{u}_{k-1}}. \tag{28}$$

In addition, we define the inverse limited memory SR1 update (see [1]) by

$$D_k = \vartheta_k I - (\vartheta_k U_k - S_k)(\vartheta_k U_k^T U_k - R_k - R_k^T + C_k)^{-1} (\vartheta_k U_k - S_k)^T, \tag{29}$$

where instead of (28) we use the value $\vartheta_k = 1$ for every k.

Next, we describe some procedures for updating the limited memory BFGS and SR1 matrices. In addition to the two $n \times m_k$ -matrices S_k and U_k, the $m_k \times m_k$ -matrices R_k, $U_k^T U_k$, and C_k are stored. Since in practice m_k is clearly smaller than n, the storage space required by these three auxiliary matrices is insignificant but the savings in computational efforts are considerable. We also give some ideas of how the search direction $\mathbf{d}_k$ can be calculated by using these different updates. After discussing the calculations separately, we then link them together.

At the k-th iteration, we have to update the limited memory representation of D_{k-1} to get D_k. Thus, we delete a column from S_{k-1} and U_{k-1} and add a new column to each of these matrices. Then, we make the corresponding updates to R_{k-1}, $U_{k-1}^T U_{k-1}$, and C_{k-1}. These updates can be done in $O(m_k^2)$ operations by storing a small amount of additional information, namely the

m_k-vectors $S_{k-1}^T\boldsymbol{\xi}_m$ and $U_{k-1}^T\boldsymbol{\xi}_m$ from the previous iteration. For example, the new triangular matrix R_k is formed from R_{k-1} (see (26)) by deleting the first row and the first column if $m_k = m_{k-1}$ and by adding a new column to the right and a new row to the bottom of the matrix. The new column is given by

$$S_k^T\mathbf{u}_{k-1} = S_k^T(\boldsymbol{\xi}_k - \boldsymbol{\xi}_m) \tag{30}$$

and the new row has the value zero in its first $m_k - 1$ components. The product $S_k^T\mathbf{u}_{k-1}$ can be computed efficiently since we already know $m_k - 1$ components of $S_k^T\boldsymbol{\xi}_m$ from $S_{k-1}^T\boldsymbol{\xi}_m$. We only need to calculate $\mathbf{s}_{k-1}^T\boldsymbol{\xi}_m$ and carry out the subtractions. The matrix $U_k^TU_k$ can be updated in a similar way. In this case, both the new column and the new row are given by $U_k^T\mathbf{u}_{k-1}$. The diagonal matrix C_k is updated by deleting the first element of C_{k-1} and adding $\mathbf{s}_{k-1}^T\mathbf{u}_{k-1}$ as the last element (note that C_k is stored as a vector).

Next, we give an efficient algorithm for updating the limited memory SR1 matrix D_k and for computing the search direction $\mathbf{d}_k = -D_k\tilde{\boldsymbol{\xi}}_k$. This algorithm is used whenever the previous step was a null step. Suppose that the number of current corrections is m_k and that we have the current iteration point $\mathbf{x}_k$, the previous corrections $\mathbf{s}_{k-1}$ and $\mathbf{u}_{k-1}$, the current (auxiliary) subgradient $\boldsymbol{\xi}_k \in \partial f(\mathbf{y}_k)$, the current aggregate subgradient $\tilde{\boldsymbol{\xi}}_k$, the basic subgradient $\boldsymbol{\xi}_m \in \partial f(\mathbf{x}_k)$, the $n \times m_k$ -matrices S_{k-1} and U_{k-1}, the $m_k \times m_k$ -matrices R_{k-1}, $U_{k-1}^TU_{k-1}$, and C_{k-1}, and the vectors $S_{k-1}^T\boldsymbol{\xi}_m$ and $U_{k-1}^T\boldsymbol{\xi}_m$ available.

Algorithm 2. (SR1 Updating and Direction Finding).

Step 1: If

$$-\mathbf{d}_{k-1}^T\mathbf{u}_{k-1} - \tilde{\boldsymbol{\xi}}_{k-1}^T\mathbf{s}_{k-1} < 0,$$

then update the matrices (i.e., go to Step 2). Otherwise, skip the updates, that is, set $S_k = S_{k-1}$, $U_k = U_{k-1}$, $R_k = R_{k-1}$, $U_k^TU_k = U_{k-1}^TU_{k-1}$, $C_k = C_{k-1}$, $S_k^T\boldsymbol{\xi}_m = S_{k-1}^T\boldsymbol{\xi}_m$, and $U_k^T\boldsymbol{\xi}_m = U_{k-1}^T\boldsymbol{\xi}_m$ and go to Step 6.

Step 2: Obtain S_k and U_k by updating S_{k-1} and U_{k-1}.

Step 3: Compute m_k-vectors $S_k^T\boldsymbol{\xi}_k$ and $U_k^T\boldsymbol{\xi}_k$.

Step 4: Compute m_k-vectors $S_k^T\mathbf{u}_{k-1}$ and $U_k^T\mathbf{u}_{k-1}$ by using the fact

$$\mathbf{u}_{k-1} = \boldsymbol{\xi}_k - \boldsymbol{\xi}_m.$$

Store m_k-vectors $S_k^T\boldsymbol{\xi}_m$ and $U_k^T\boldsymbol{\xi}_m$.

Step 5: Update $m_k \times m_k$ -matrices R_k, $U_k^T U_k$, and C_k.

Step 6: Set $\vartheta_k = 1.0$.

Step 7: Compute m_k-vectors $S_k^T \tilde{\boldsymbol{\xi}}_k$ and $U_k^T \tilde{\boldsymbol{\xi}}_k$.

Step 8: Compute

$$\mathbf{p} = (\vartheta_k U_k^T U_k - R_k - R_k^T + C_k)^{-1}(\vartheta_k U_k^T \tilde{\boldsymbol{\xi}}_k - S_k^T \tilde{\boldsymbol{\xi}}_k).$$

Step 9: Compute

$$\mathbf{d}_k = -\vartheta_k \tilde{\boldsymbol{\xi}}_k + (\vartheta_k U_k - S_k)\mathbf{p}.$$

Note that the condition (see Step 1)

$$-\mathbf{d}_i^T \mathbf{u}_i - \tilde{\boldsymbol{\xi}}_i^T \mathbf{s}_i < 0 \qquad \text{for all } i = 1, \ldots, k-1 \tag{31}$$

assures the positive definiteness of the matrices obtained by the limited memory SR1 update (see [6]).

Due to the fact that after a serious step the aggregate subgradient $\tilde{\boldsymbol{\xi}}_k = \boldsymbol{\xi}_k \in \partial f(\mathbf{x}_k)$ and

$$\begin{aligned} \mathbf{s}_k &= \mathbf{x}_{k+1} - \mathbf{x}_k \qquad \text{and} \\ \mathbf{u}_k &= \boldsymbol{\xi}_{k+1} - \boldsymbol{\xi}_k, \end{aligned}$$

(note that in the case of a serious step this representation of $\mathbf{s}_k$ and $\mathbf{u}_k$ is not conflicting with (23)), the calculations used are very similar to those given in [1]. In fact, all the calculations in [1] could be done by replacing the gradient $\nabla f(\mathbf{x})$ by an arbitrary subgradient $\boldsymbol{\xi} \in \partial f(\mathbf{x})$. However, rather than updating and inverting the upper triangular matrix R_k at every iteration, we update and store the inverse R_k^{-1}.

We now give an efficient algorithm for updating the limited memory BFGS matrix D_k and for computing the search direction $\mathbf{d}_k = -D_k \boldsymbol{\xi}_k$ when the previous step was a serious step. Suppose that we have the current subgradient $\boldsymbol{\xi}_k \in \partial f(\mathbf{x}_k)$, the previous subgradient $\boldsymbol{\xi}_{k-1} \in \partial f(\mathbf{x}_{k-1})$, the $m_k \times m_k$ -matrices R_{k-1}^{-1}, $U_{k-1}^T U_{k-1}$, and C_{k-1}, and the previous multiplier ϑ_{k-1} available.

Algorithm 3. (BFGS Updating and Direction Finding).

Step 1: If

$$\mathbf{u}_{k-1}^T \mathbf{s}_{k-1} > 0,$$

then update the matrices (i.e., go to Step 2). Otherwise, skip the updates, that is, set $S_k = S_{k-1}$, $U_k = U_{k-1}$, $R_k = R_{k-1}$, $U_k^T U_k = U_{k-1}^T U_{k-1}$, $C_k = C_{k-1}$, and $\vartheta_k = \vartheta_{k-1}$, compute $S_k^T \boldsymbol{\xi}_k$ and $U_k^T \boldsymbol{\xi}_k$, and go to Step 7.

Step 2: Obtain S_k and U_k by updating S_{k-1} and U_{k-1}.

Step 3: Compute and store m_k-vectors $S_k^T \boldsymbol{\xi}_k$ and $U_k^T \boldsymbol{\xi}_k$.

Step 4: Compute m_k-vectors $S_k^T \mathbf{u}_{k-1}$ and $U_k^T \mathbf{u}_{k-1}$ by using the fact

$$\mathbf{u}_{k-1} = \boldsymbol{\xi}_k - \boldsymbol{\xi}_{k-1}.$$

Step 5: Update $m_k \times m_k$ -matrices R_k^{-1}, $U_k^T U_k$, and C_k.

Step 6: If $\mathbf{u}_{k-1}^T \mathbf{u}_{k-1} > 0$, compute ϑ_k

$$\vartheta_k = \frac{\mathbf{u}_{k-1}^T \mathbf{s}_{k-1}}{\mathbf{u}_{k-1}^T \mathbf{u}_{k-1}}.$$

Note that both $\mathbf{u}_{k-1}^T \mathbf{s}_{k-1}$ and $\mathbf{u}_{k-1}^T \mathbf{u}_{k-1}$ have already been calculated. Otherwise, set $\vartheta_k = 1.0$.

Step 7: Compute two intermediate values

$$\begin{aligned} \mathbf{p}_1 &= R_k^{-1} S_k^T \boldsymbol{\xi}_k, \\ \mathbf{p}_2 &= (R_k^{-1})^T (C_k \mathbf{p}_1 + \vartheta_k U_k^T U_k \mathbf{p}_1 - \vartheta_k U_k^T \boldsymbol{\xi}_k). \end{aligned}$$

Step 8: Compute

$$\mathbf{d}_k = \vartheta_k U_k \mathbf{p}_1 - S_k \mathbf{p}_2 - \vartheta_k \boldsymbol{\xi}_k.$$

Note that the condition (see Step 1)

$$\mathbf{u}_i^T \mathbf{s}_i > 0 \qquad \text{for all } i = 1, \ldots, k-1 \tag{32}$$

assures the positive definiteness of the matrices obtained by the limited memory BFGS update (see, e.g., [1]).

In order to use both the Algorithms 2 and 3 with the same stored information, some modifications have to be made. Firstly, we have to update and store both matrices R_k and R_k^{-1} at each iteration regardless of the update formula we are using. In addition, since we use the same correction matrices S_k and U_k for the calculations of both the BFGS and the SR1 updates, we have to test both the positive definiteness conditions (31) and (32) in each case before we update the matrices. However, numerical experiments have showed that the simple skipping of the updates (see Algorithms 2 and 3, Step 1), if both the required conditions are not satisfied, makes the method quite inefficient. This is due to the fact that the BFGS update was usually skipped due to condition (31) required for the SR1 update. Therefore, we use the most recent corrections $\mathbf{s}_{k-1}$ and $\mathbf{u}_{k-1}$ to calculate the new search direction $\mathbf{d}_k$ whenever the required positive definiteness condition is valid but the matrices are not updated unless both the conditions (31) and (32) are satisfied. In practice, this means that the correction matrices S_k and U_k may actually include some indices smaller than $k - m_k$ and that the number of the current corrections used may be $m_k = m_c + 1$.

The new limited memory bundle method uses a limited memory approach to calculate the search direction and it requires only three subgradients and two locality measures to calculate the new aggregate values. Thus, the time-consuming quadratic subproblem (6) appearing in standard bundle methods needs not to be solved and the size of the bundle needs not to increase with the dimension of the problem. Furthermore, both the search direction $\mathbf{d}_k$ and the aggregate values $\tilde{\boldsymbol{\xi}}_{k+1}$ and $\tilde{\beta}_{k+1}$ can be computed implicitly using at most $O(nm_c)$ operations. Assuming $m_c \ll n$, this is much less than $O(n^2)$ operations needed with the original variable metric bundle method, which stores and manipulates the whole matrix D_k. These improvements make the limited memory bundle method suitable for large-scale problems. This assertion is supported by numerical tests presented in Section 4.

3 Large-Scale Nonsmooth Test Problems

Many practical optimization applications involve nonsmooth functions of many variables. However, there exist only few large-scale academic test problems for the nonsmooth case. For this reason, we now introduce a new set of large-scale unconstrained minimization problems for nonsmooth optimization.

We have made 10 nonsmooth problems which all can be formulated with any number of variables. The problems are constructed either by chaining and extending small existing nonsmooth problems or by "nonsmoothing" large smooth problems (that is, for example, by replacing the term x_i^2 with $|x_i|$). First, we give the formulation of the objective function $f(\mathbf{x})$ and the starting point $\mathbf{x}_1$ for each problem. Then, we collect some details of the problems as well as the references to the original problems in Table 1.

1. Generalization of MAXQ

$$f(\mathbf{x}) = \max_{1 \le i \le n} x_i^2.$$

$$\begin{aligned} x_i^1 &= i, && \text{for } i = 1, \ldots, n/2 \text{ and} \\ x_i^1 &= -i, && \text{for } i = n/2 + 1, \ldots, n. \end{aligned}$$

2. Generalization of MXHILB

$$f(\mathbf{x}) = \max_{1 \le i \le n} \left| \sum_{j=1}^{n} \frac{x_j}{i+j-1} \right|.$$

$$x_i^1 = 1, \qquad \text{for all } i = 1, \ldots, n.$$

3. Chained LQ

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} \max \left\{ -x_i - x_{i+1}, -x_i - x_{i+1} + (x_i^2 + x_{i+1}^2 - 1) \right\}.$$

$$x_i^1 = -0.5, \quad \text{for all } i = 1, \ldots, n.$$

4. Chained CB3 I

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} \max \left\{ x_i^4 + x_{i+1}^2, (2 - x_i)^2 + (2 - x_{i+1})^2, 2e^{-x_i + x_{i+1}} \right\}.$$

$$x_i^1 = 2, \qquad \text{for all } i = 1, \ldots, n.$$

5. Chained CB3 II

$$\begin{aligned} f(\mathbf{x}) = \max \Big\{ & \sum_{i=1}^{n-1} \left(x_i^4 + x_{i+1}^2 \right), \sum_{i=1}^{n-1} \left((2 - x_i)^2 + (2 - x_{i+1})^2 \right), \\ & \sum_{i=1}^{n-1} \left(2e^{-x_i + x_{i+1}} \right) \Big\}. \end{aligned}$$

$$x_i^1 = 2, \qquad \text{for all } i = 1, \ldots, n.$$

6. Number of Active Faces

$$f(\mathbf{x}) = \max_{1 \le i \le n} \{\, g\left(-\sum_{i=1}^{n} x_i\right), g(x_i) \,\},$$

where $g(y) = \ln(|y| + 1)$.
$x_i^1 = 1, \qquad$ for all $i = 1, \ldots, n$.

7. Nonsmooth generalization of Brown function 2

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} \left(|x_i|^{x_{i+1}^2+1} + |x_{i+1}|^{x_i^2+1} \right).$$

$x_i^1 = -1, \qquad$ when $\mod(i, 2) = 1$, $(i = 1, \ldots, n)$ and
$x_i^1 = 1, \qquad$ when $\mod(i, 2) = 0$, $(i = 1, \ldots, n)$.

8. Chained Mifflin 2

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} \left(-x_i + 2\left(x_i^2 + x_{i+1}^2 - 1\right) + 1.75\left|x_i^2 + x_{i+1}^2 - 1\right| \right).$$

$x_i^1 = -1, \qquad$ for all $i = 1, \ldots, n$.

9. Chained Crescent I

$$f(\mathbf{x}) = \max \Big\{ \sum_{i=1}^{n-1} \left(x_i^2 + (x_{i+1} - 1)^2 + x_{i+1} - 1\right), \sum_{i=1}^{n-1} \left(-x_i^2 - (x_{i+1} - 1)^2 + x_{i+1} + 1\right) \Big\}.$$

$x_i^1 = -1.5, \quad$ when $\mod(i, 2) = 1$, $(i = 1, \ldots, n)$ and
$x_i^1 = 2.0, \quad$ when $\mod(i, 2) = 0$, $(i = 1, \ldots, n)$.

10. Chained Crescent II

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} \max \left\{ x_i^2 + (x_{i+1} - 1)^2 + x_{i+1} - 1, -x_i^2 - (x_{i+1} - 1)^2 + x_{i+1} + 1 \right\}.$$

$x_i^1 = -1.5, \quad$ when $\mod(i, 2) = 1$, $(i = 1, \ldots, n)$ and
$x_i^1 = 2.0, \quad$ when $\mod(i, 2) = 0$, $(i = 1, \ldots, n)$.

The details of the problems are given in Table 1, where P denotes the problem number, $f(\mathbf{x}^*)$ is the minimum value of the objective function, and the

Table 1: Test problems

P	$f(\mathbf{x}^*)$	Convex	Original problem and reference
1	0.0	+	MAXQ, $n = 20$, see, e.g., [20]
2	0.0	+	MXHILB, $n = 50$, see, e.g., [18]
3	$-(n-1)\sqrt{2}$	+	LQ, $n = 2$, see, e.g., [20]
4	$2(n-1)$	+	CB3, $n = 2$, see, e.g., [20]
5	$2(n-1)$	+	CB3, $n = 2$, see, e.g., [20]
6	0.0	−	See [5]
7	0.0	−	Generalization of Brown function 2, see, e.g., [15]
8	varies*	−	Mifflin 2, $n = 2$, see, e.g., [20]
9	0.0	−	Crescent, $n = 2$, see, e.g., [20]
10	0.0	−	Crescent, $n = 2$, see, e.g., [20]

* $f(\mathbf{x}^*) \approx -6.51$ for $n = 10$, $f(\mathbf{x}^*) \approx -70.15$ for $n = 100$ and $f(\mathbf{x}^*) \approx -706.55$ for $n = 1000$.

symbols − (nonconvex) and + (convex) denote the convexity of the problems. Also the references to the original problem in each case are given in Table 1.

4 Numerical Experiments

In order to get some information of how the new method works in practice when compared to other nonsmooth methods, we tested different existing programs with several problems. In this section, we first introduce the tested software. Then, we give the results from the numerical experiments and draw some conclusions.

4.1 Tested Software

In this subsection, we first introduce the programs used in our experiments. The experiments were performed in a SGI Origin 2000/128 supercomputer (MIPS R12000, 600 Mflop/s/processor). The algorithms were implemented in FORTRAN77 with double-precision arithmetic. The pieces of software tested are presented in Table 2. None of the nonsmooth optimization programs `PVAR`, `PBUN`, `PNEW`, and `PBNCGC` has been developed for large-scale optimization. On the other hand, we wanted to get some information of the behavior of the new program `LMBM` especially with large-scale problems. For this reason, we used the smooth large-scale optimization program `L-BFGS` as a benchmark. Thus, all the programs were first tested with 22 smooth

Table 2: Tested pieces of software

Software	Author(s)	Method	Reference
PVAR	Lukšan & Vlček	Variable metric bundle	[14, 24]
PNEW	Lukšan & Vlček	Bundle-Newton	[13]
PBUN	Lukšan & Vlček	Proximal bundle	[17]
PBNCGC	Mäkelä	Proximal bundle	[20]
L-BFGS	Nocedal	Limited memory BFGS	[12, 21]
LMBM	Haarala	Limited memory bundle	

minimization problems, which all could be formulated with any number of variables. A detailed description of these problems can be found in [15]. Then, the programs for nonsmooth optimization (that is, all the programs in Table 2 except L-BFGS) were tested with 10 nonsmooth minimization problems described earlier. Finally, the programs for nonsmooth optimization were tested with a practical nonsmooth image restoration problem. A detailed description of the problem can be found in [10].

4.2 Numerical Results

Smooth test problems. All the programs given in Table 2 were first tested with 22 smooth problems with the numbers of variables 10, 100 and 1000 and in case of the limited memory programs L-BFGS and LMBM also with 10 000 variables.

We tested the bundle programs PVAR, PNEW, PBUN, PBNCGC, and LMBM with relatively small sizes of the bundle (m_ξ). That is, $m_\xi = 10$ for the bundle-Newton program PNEW and for both the proximal bundle programs PBUN and PBNCGC, and $m_\xi = 2$ for the variable metric bundle programs PVAR and LMBM. For the limited memory programs L-BFGS and LMBM, the maximum number of stored corrections (m_c) was set to 7. As a stopping parameter, we used $\varepsilon = 10^{-6}$ in all the cases. The other parameters used were chosen experimentally.

In the test results to be reported, we say that the optimization terminated successfully if

- the problem was solved with the desired accuracy. That is, $w_k \leq \varepsilon$, where $w_k = \frac{1}{2}\|\tilde{\boldsymbol{\xi}}_k\|^2 + \tilde{\beta}_k$ in PNEW, PBUN, and PBNCGC, $w_k = \tilde{\boldsymbol{\xi}}_k^T D_k \tilde{\boldsymbol{\xi}}_k + 2\tilde{\beta}_k$ in PVAR, $w_k = 2\,\tilde{\boldsymbol{\xi}}_k^T D_k \tilde{\boldsymbol{\xi}}_k + 4\tilde{\beta}_k$ in LMBM (note that also $\frac{1}{2}\|\tilde{\boldsymbol{\xi}}_k\|^2 + \tilde{\beta}_k \leq 10^3\varepsilon$), and $w_k = \|\nabla f(\mathbf{x}_k)\| / \max\{1, \|\mathbf{x}_k\|\}$ in L-BFGS.

In addition, for the programs `PVAR`, `PNEW`, `PBUN` and `LMBM` we say that the optimization terminated successfully if

- $|f_{k+1} - f_k| \leq 1.0 \cdot 10^{-8}$ in 10 subsequent iterations.

Otherwise, we say that the optimization failed.

The results of the smooth experiments are summarized in Figure 1 and in Table 3. In Figure 1, we give the average CPU time elapsed for problems in proportion to the number of variables for each of the six programs (note that we have naturally calculated some extra data points to obtain a realistic figure). In Table 3, we have calculated the average number of iterations (Ni) and function evaluations (Nf) needed for problems of different sizes. The problems where the optimization has failed (fail) are not included in the data. Since for almost every tested program there existed one problem in the set of 22 problems that needed much more iterations than the others, we also removed in each case the problems that used the largest and the smallest number of iterations. In addition, we give the number of the inaccurate results occurred (#) within the 22 problems. That is, the number of problems where optimization has not failed but the precision of the result is more than two significant digits greater than the desired accuracy of the solution. The blanks in the table mean that the problems were not tested in these cases.

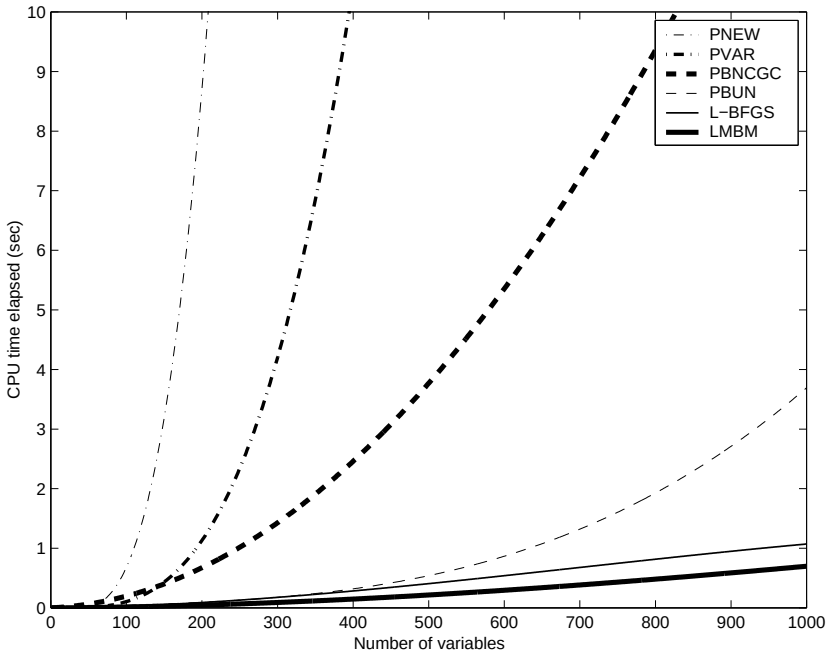

Figure 1: Average CPU time elapsed for smooth problems.

Table 3: Average results with 22 smooth problems.

Program/n	10	100	1000	10000
	Ni/Nf/fail	Ni/Nf/fail	Ni/Nf/fail	Ni/Nf/fail
PVAR	34/35/0	170/179/0	1668/1885/$0^{(1)}$	–
PNEW	20/21/0	35/38/0	20/21/3	–
PBUN	71/73/0	216/218/$1^{(4)}$	1407/1410/$1^{(7)}$	–
PBNCGC	80/99/0	338/428/0	2414/4131/0	–
L-BFGS	36/45/0	181/212/1	960/998/3	13142/14522/$5^{(1)}$
LMBM	31/34/0	92/105/0	338/355/0	6432/7587/$0^{(1)}$

(#) Number (#) of inaccurate results obtained.

To sum up, all the tested programs worked well for small- and medium-scale problems ($n \leq 100$) but with large-scale problems the programs other than those using the limited memory approach became computationally inefficient (see Figure 1). The new limited memory bundle program LMBM was the most efficient method in all the cases. It found the local minimum in a reliable way and in our experiments it was also very robust while, for example, PVAR and PBUN were very sensitive to the choice of internal parameters. Also the limited memory BFGS program L-BFGS was efficient with both small- and large-scale smooth problems. However, when the dimension of the problems increased, L-BFGS had some difficulties with the line search and it failed to solve some of the problems.

Nonsmooth test problems. The programs for nonsmooth optimization (that is, all the programs in Table 2 except L-BFGS) were tested also with the ten nonsmooth minimization problems 1–10 described in Section 3. The numbers of variables used were 10, 100 and 1000.

We tested the programs with different sizes of bundles (m_ξ). For the bundle-Newton program PNEW and for both the proximal bundle programs PBUN and PBNCGC, the sizes of bundles used were 10 and 100, and for the variable metric bundle programs PVAR and LMBM, the sizes of bundles were 2 and 100. For LMBM, the maximum number of stored corrections (m_c) was set to 7.

In our nonsmooth experiments, the following values of parameters were used:

- For convex problems 1–5, the distance measure parameter $\gamma = 0$ was used with the programs PBUN, PBNCGC, and LMBM.

  With PNEW, the distance measure parameter $\gamma = 0.001$ was used since the value of γ has to be positive (see [13]).

  With PVAR, the convex version of the program was used.

- For nonconvex problems 6–10, the distance measure parameter $\gamma = 0.50$ was used with all the programs and we used the nonconvex version of the program `PVAR`.
- The stopping parameter $\varepsilon = 10^{-5}$ was used in all the cases.
- Otherwise, the default parameters were used with all the programs.
- In addition to the stopping criteria already mentioned, we terminated the experiments if the CPU time elapsed exceeded half an hour.

The average results of the nonsmooth experiments are summarized in Table 4. For all the programs, we first give the results obtained with the smaller bundle and then below the results obtained with the larger bundle. The average CPU time elapsed (time) is given in seconds. Otherwise, the results are given as before and in each case we have removed from the data the problems that used the largest and the smallest number of iterations.

Table 4: Average results with 10 nonsmooth problems.

Program/n	10		100		1000	
	Ni/Nf/fail	time	Ni/Nf/fail	time	Ni/Nf/fail	time
`PVAR`	185/200/1	0.006	589/597/2	0.29	$17782/6497/3^{(3)}$	902.61
	81/86/0	0.005	303/311/1	0.26	$1713/1717/3^{(1)}$	156.61
`PNEW`	80/89/1	0.014	296/314/3	3.26	$164/273/1^{(2)}$	1324.12
	90/98/0	0.084	144/158/2	8.33	$73/146/1^{(2)}$	957.61
`PBUN`	101/105/0	0.007	840/918/1	0.30	913/1031/4	10.17
	56/59/0	0.007	222/230/2	0.73	1958/2010/1	46.13
`PBNCGC`	55/64/0	0.005	16799/16957/3	7.94	$139526/139545/0^{(1)}$	906.20
	41/50/0	0.005	269/307/1	2.61	3337/3348/1	771.85
`LMBM`	583/3113/0	0.031	373/1176/0	0.08	420/1374/0	0.76
	126/152/0	0.008	$238/269/0^{(1)}$	0.10	$232/283/1^{(1)}$	1.38

(#) Number (#) of inaccurate results obtained.

For smooth problems, the average results (see Table 3 and Figure 1) give quite a realistic impression of the behavior of the programs. However, for nonsmooth problems, the average results given in Table 4 may be misleading especially when the size of the problems is large. For example, with 1000 variables, the proximal bundle program `PBNCGC` either used the whole time limit available (in problems 1, 3, 4, 8 and 10) or then solved the problem really fast (in problems 2, 5, 6, 7 and 9). In fact, `PBNCGC` (with $m_\xi = 100$) was the most efficient program tested with three problems (that is, problems 2, 6 and 7). The same kind of an effect can also be seen with the other

proximal bundle program PBUN, although it never needed the whole time limit and, on the other hand, it was the most efficient program only with one problem (in problem 1). PBUN was also the only program that could solve problem 1 with 1000 variables properly within the given time limit. With all the other problems our new program LMBM, was the most efficient method with 1000 variables. In fact, LMBM was usually the most efficient program also with 100 variables but the differences were not substantial in these cases. In all the cases, LMBM outperformed the original variable metric bundle program PVAR and the bundle Newton program PNEW already with 100 variables.

With large-scale problems, our new program LMBM usually needed less iterations and function evaluations than the other programs except the bundle Newton program PNEW which, however, was the most time-consuming of the programs tested due to matrix operations. Thus, LMBM should be an efficient method also in the cases where the function and the subgradient evaluations are expensive.

Now, let us for a while concentrate only on our new program LMBM since in large-scale cases it usually outperforms the other bundle programs tested. We tested LMBM with different sizes of bundles and with different maximum numbers of stored corrections. The sizes of the bundles were the same as before, that is $m_\xi = 2$ and $m_\xi = 100$, and the maximum numbers of stored corrections were set to 3, 7, and 15. In what follows, we denote these different modifications by LMBM(3), LMBM(7), and LMBM(15).

In Table 5, we report the results obtained with the different modifications of our new program for each nonsmooth problem with 1000 variables. As before, we first give the results obtained with the smaller bundle and then below the results obtained with the larger bundle. At the bottom of Table 5, we give the average results calculated exactly as in Table 4. Our goal is to identify the classes of problems for which our new program is effective. In addition, we are interested in the best values for the maximum number of stored corrections and the size of the bundle.

It can be seen in Table 5 that our new program had serious difficulties with problem 1. These difficulties were quite predictable, since there exists only one nonzero component in the subgradient vector of the objective function at each iteration. In practice, this means that the approximation of the inverse of the Hessian matrix becomes sparse, and thus, the search direction may be quite inaccurate. Also smooth limited memory methods have been reported to be best suited for problems where the Hessian matrix is not very sparse [12].

Table 5: Results with the nonsmooth problems with 1000 variables.

	LMBM(3)		LMBM(7)		LMBM(15)	
P	Ni/Nf	Time	Ni/Nf	Time	Ni/Nf	Time
1	2390162/2390383*	1800.00	1627489/2098714*	1800.00	914774/915235*	1800.00
	fail	–	fail	–	fail	–
2	326/1007*	70.78	61/135*	9.49	104/263*	18.49
	322/417*	30.00	62/102*	7.28	63/104*	7.36
3	167/528	0.12	152/422	0.13	223/587	0.27
	138/163	0.36	153/195	0.44	203/303	0.77
4	158/483	0.22	250/820	0.43	263/825	0.54
	32/33	0.03	125/159	0.35	219/331	0.96
5	32/87	0.04	150/424	0.22	169/372	0.22
	32/38	0.04	81/89	0.14	58/60	0.09
6	536/537	0.22	538/539	0.22	538/539	0.23
	529/531	2.16	529/531	2.13	529/531	2.13
7	283/1179	2.19	417/1672	3.33	237/431	0.99
	237/244*	1.17	168/183	2.56	183/224	0.92
8	416/1319	0.28	710/2462	0.64	1351/3315	1.62
	230/308	0.79	479/650	2.08	1749/2192	9.00
9	149/351	0.10	92/103	0.06	142/238	0.11
	159/162	0.52	143/143	0.38	127/268	0.30
10	774/4312*	0.71	1049/4509	1.07	867/3062	1.17
	276/342*	1.00	477/559*	2.73	366/467*	1.64
Aver.	351/1215	9.33	420/1374	0.76	474/1171	0.64
Aver.	199/239	4.84	232/283	1.38	241/318	2.01

* An inaccurate result obtained.

The program LMBM had also some difficulties with problem 2. With all the tested versions, the optimization was terminated before the minimum point was actually achieved. The reason for this premature termination is that at every iteration, the subgradient vector of the objective function is of the form $\pm(1/i, 1/(i+1), \ldots, 1/(i+n-1))$, where i is the index of the max-function. When i is large, the norm of the subgradient becomes small and we stop the computation. For some reason, the approximation of the inverse of the Hessian matrix does not prevent this termination. Anyhow, this kind of a difficulty is very easy to avoid: we just have to tighten the second stopping criterion in (22).

All the other problems 3–10 were solved successfully with our new program LMBM. In all these cases, the subgradient vector of the objective function contains many nonzero entries and the values of these components depend on the current iteration point $\mathbf{x}_k$. However, also with these problems, there occurred some inaccurate results especially with the version LMBM(3). The computational times used with LMBM(3) were usually little smaller than those with LMBM(7) or LMBM(15) but the differences were insignificant. When comparing the versions LMBM(7) and LMBM(15), there was no a substantial difference in

the accuracy of the program. Thus, we can say that the maximum number of stored corrections should be at least 7.

The numbers of iterations and function evaluations needed with `LMBM` were usually significantly smaller when the size of the bundle was large. This is due to the fact that the selection of the initial step size is more accurate when a larger bundle is used. On the other hand, each individual iteration was more costly when the size of the bundle was increased. In practice, this means that for problems with expensive objective function and subgradient evaluations, it is better to use larger bundles, and thus, fewer iterations and function evaluations.

We conclude from these experiments that our new method is best suited for problems with dense subgradient vectors where components depend on the current iteration point $\mathbf{x}_k$ (that is, the components are not constants). In addition, we conclude that the maximum number of stored corrections should be at least 7.

Image restoration problem. Finally, we tested the nonsmooth optimization programs with a convex image restoration problem (see [10]). Since for problems 1–10, the results of our new program with a small maximum number of stored corrections ($m_c = 3$) were quite inaccurate, we only tested the image restoration problem with $m_c = 7$ and $m_c = 15$. The stopping parameter $\varepsilon = 10^{-4}$ was used with all the programs. Otherwise, the parameters similar to the convex problems 1–5 were used.

In Figures 2 and 3, we give the CPU times elapsed with the problem with a different number of variables. In addition, we give some more specified results for the problem with 100, 500, and 1000 variables in Table 6, where f denotes the value of the objective function at termination.

From the numerical results, we can conclude the superiority of the limited memory bundle program `LMBM` when comparing the computational times (see Figures 2 and 3). In all the cases, it used significantly less CPU time than the other programs. However, the accuracy of the new program was somewhat disappointing. The minima of the objective function found with `LMBM` were usually a little bit greater than with the other programs (especially those found with the proximal bundle programs `PBNCGC` and `PBUN`). In all the cases, the result obtained with our new program `LMBM` became more accurate when the maximum number of stored corrections or the size of the bundle was increased (see Table 6).

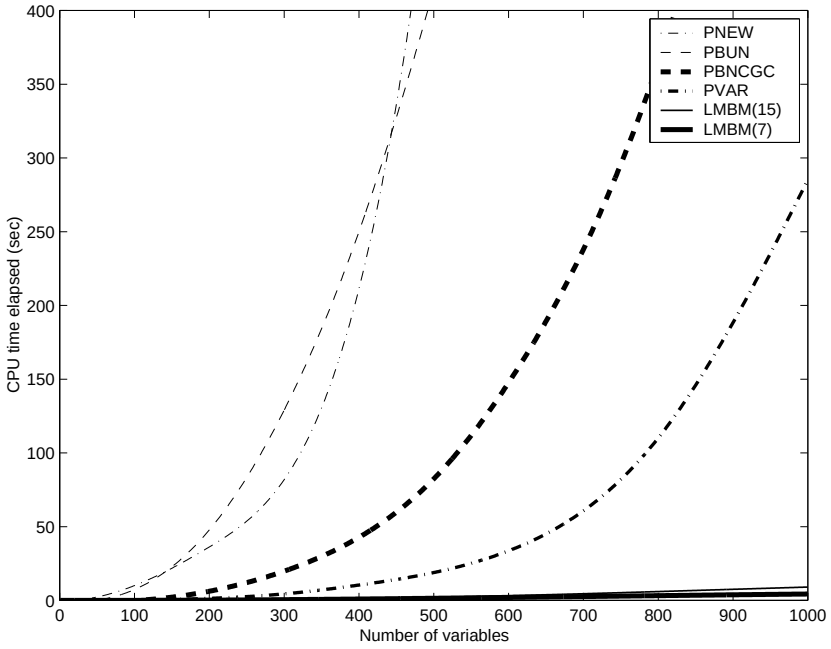

Figure 2: CPU time elapsed for the problem with small bundles.

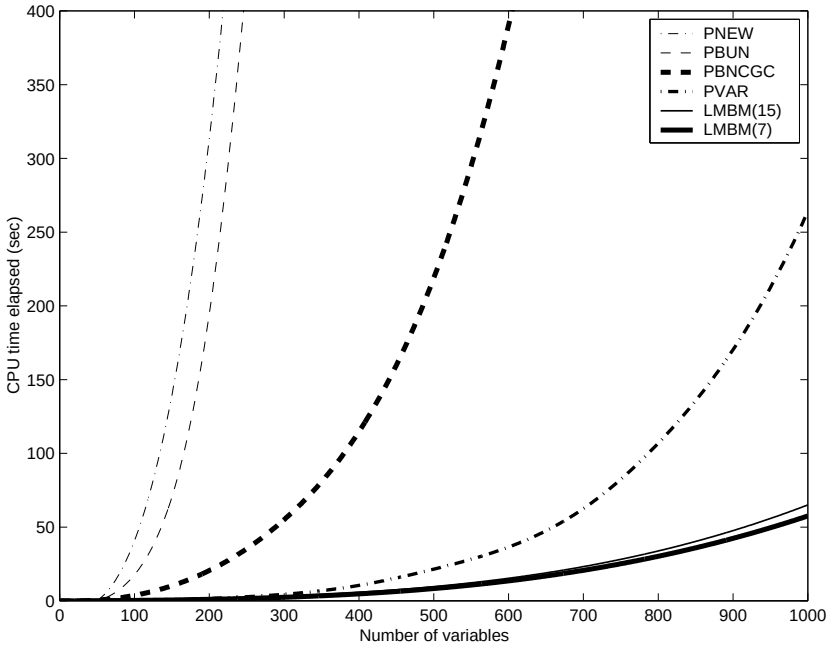

Figure 3: CPU time elapsed for the problem with large bundles.

Table 6: Results for the image restoration problem.

Program/n	100		500		1000	
	Ni/Nf	f	Ni/Nf	f	Ni/Nf	f
PVAR	370/370	0.79751	1787/1787	4.87495	3281/3281	9.86366
	266/266	0.79750	1689/1689	4.86216	2932/2932	9.80222
PNEW	993/1138	0.79752	998/1185	4.87698	267/293	11.39220
	242/250	0.79750	773/861	4.87243	110/121	12.36375
PBUN	45232/51461	0.79750	196206/233754	4.86189	377092/480182	9.76172
	2281/2310	0.79750	71081/79933	4.86200	43136/48470	9.76818
PBNCGC	5900/5901	0.79751	25568/25569	4.86194	116058/116059	9.76172
	300/301	0.79751	3590/3591	4.86194	9913/9914	9.76184
LMBM(7)	392/821	0.79845	2280/2545	4.89094	5792/6121	9.82068
	643/710	0.79827	3486/3524	4.87367	11642/11677	9.80511
LMBM(15)	333/779	0.79824	3543/4341	4.87126	8534/9133	9.80616
	502/608	0.79823	5388/5522	4.86814	12148/12246	9.78286

Note that the results obtained with this practical problem differ a lot from those obtained before with smooth problems and with nonsmooth problems 1–10 especially with the program PBUN. In all the cases, PBUN used much more iterations and function evaluations than the other proximal bundle program PBNCGC (see Table 6). It was also very time-consuming (see Figures 2 and 3) while with smooth problems and with problems 1–10 it was the most efficient of the current bundle programs tested. On the other hand, the variable metric bundle program PVAR was the most efficient of the current bundle programs, while with academic problems it was very time-consuming.

Also LMBM behaved a little differently with the practical problem than before: In each case it used more iterations and function evaluations when the larger bundle was used. However, as said before, also the results obtained with LMBM became more accurate when the size of the bundle was increased.

We conclude from these experiments that our new method was usually the most efficient method for large-scale problems. With smooth problems, the new limited memory bundle program LMBM was almost twice as fast as the limited memory variable metric program L-BFGS that has been developed for smooth large-scale minimization. In addition, for example, with 1000 variables, LMBM was on an average about 5 times faster than the fastest bundle program PBUN and 250 times faster than the original variable metric bundle program PVAR. For nonsmooth problems these differences were even more perceptible. For example, for the image restoration problem with 500 variables, LMBM was about 30 times faster than PVAR, 100 times faster than the proximal bundle program PBNCGC, 450 times faster than the other proximal bundle program PBUN and 750 times faster than the bundle-Newton program PNEW.

5 Conclusions

In this paper, we have introduced a new limited memory bundle method for nonsmooth large-scale optimization. We have also tested the performance of this new method with different minimization problems. The numerical experiments confirm that the new method is efficient and reliable for both smooth and nonsmooth optimization problems. With large numbers of variables it usually used significantly less CPU time than the other programs tested.

With smooth problems, the accuracy of the new program was comparable to the other programs tested. However, with nonsmooth problems, the minima found with the limited memory bundle program were often slightly greater than those of the other programs and there occurred some inaccurate results especially with a small maximum number of stored corrections. Thus, we conclude that the maximum number of stored corrections should be at least 7.

Our numerical experiments showed that the limited memory bundle method works well for both convex and nonconvex minimization problems. Yet, it is best suited for problems with dense subgradient vectors where components depend on the current iteration point.

Although the new method is quite useful already, there is a lot of further work required before the idea is complete. Possible areas of future development include the following: alternative ways of scaling the updates (especially, the SR1 update), constraint handling (simple bounds, linear constraints, nonlinear constraints), and parallelized version of the program.

Acknowledgements

We would like to thank Dr. Ladislav Lukšan and Dr. Jan Vlček for the permission to use and modify their variable metric bundle software to make the method suitable for large-scale optimization.

This work was financially supported by COMAS Graduate School of the University of Jyväskylä, TEKES and Academy of Finland grant #65760.

References

[1] R. H. Byrd, J. Nocedal, and R. B. Schnabel. Representations of quasi-Newton matrices and their use in limited memory methods. *Mathematical Programming*, 63:129–156, 1994.

[2] F. H. Clarke. *Optimization and Nonsmooth Analysis*. Wiley-Interscience, New York, 1983.

[3] R. Fletcher. *Practical Methods of Optimization*. John Wiley and Sons, Chichester, second edition, 1987.

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

[5] A. Grothey. *Decomposition Methods for Nonlinear Nonconvex Optimization Problems*. PhD Thesis, University of Edinburgh, 2001.

[6] M. Haarala. *Bundle Methods for Large-Scale Nonsmooth Optimization*. PhLic Thesis, University of Jyväskylä, Department of Mathematical Information Technology, 2003.

[7] J.-B. Hiriart-Urruty and C. Lemaréchal. *Convex Analysis and Minimization Algorithms II*. Springer-Verlag, Berlin, 1993.

[8] K. C. Kiwiel. *Methods of Descent for Nondifferentiable Optimization*. Lecture Notes in Mathematics 1133. Springer-Verlag, Berlin, 1985.

[9] T. G. Kolda, D. P. O'Leary, and L. Nazareth. BFGS with update skipping and varying memory. *SIAM Journal of Optimization*, 8(4):1060–1083, 1998.

[10] T. Kärkkäinen, K. Majava, and M. M. Mäkelä. Comparison of formulations and solution methods for image restoration problems. *Inverse Problems*, 17(6):1977–1995, 2001.

[11] C. Lemaréchal. Nondifferentiable optimization. In G. L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd, editors, *Optimization*, pages 529–572. North-Holland, Amsterdam, 1989.

[12] D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45:503–528, 1989.

[13] L. Lukšan and J. Vlček. A bundle-Newton method for nonsmooth unconstrained minimization. *Mathematical Programming*, 83:373–391, 1998.

[14] L. Lukšan and J. Vlček. Globally convergent variable metric method for convex nonsmooth unconstrained minimization. *Journal of Optimization Theory and Applications*, 102:593–613, 1999.

[15] L. Lukšan and J. Vlček. Sparse and partially separable test problems for unconstrained and equality constrained optimization. Technical Report 767, Institute of Computer Science, Academy of Sciences of the Czech Republic, Prague, 1999.

[16] L. Lukšan and J. Vlček. Introduction to nonsmooth analysis. Theory and algorithms. Technical Report DMSIA 1/2000, University of Bergamo, 2000.

[17] L. Lukšan and J. Vlček. NDA: Algorithms for nondifferentiable optimization. Technical Report 797, Institute of Computer Science, Academy of Sciences of the Czech Republic, Prague, 2000.

[18] L. Lukšan and J. Vlček. Test problems for nonsmooth unconstrained and linearly constrained optimization. Technical Report 798, Institute of Computer Science, Academy of Sciences of the Czech Republic, Prague, 2000.

[19] M. M. Mäkelä. Survey of bundle methods for nonsmooth optimization. *Optimization Methods and Software*, 17(1):1–29, 2002.

[20] M. M. Mäkelä and P. Neittaanmäki. *Nonsmooth Optimization: Analysis and Algorithms with Applications to Optimal Control*. World Scientific Publishing Co., Singapore, 1992.

[21] J. Nocedal. Updating quasi-Newton matrices with limited storage. *Mathematics of Computation*, 35(151):773–782, 1980.

[22] N. Z. Shor. *Minimization Methods for Non-Differentiable Functions*. Springer-Verlag, Berlin, 1985.

[23] P. L. Toint. On sparse and symmetric matrix updating subject to a linear equation. *Mathematics of Computation*, 31(140):954–961, 1977.

[24] J. Vlček and L. Lukšan. Globally convergent variable metric method for nonconvex nondifferentiable unconstrained minimization. *Journal of Optimization Theory and Applications*, 111(2):407–430, 2001.