Solving Nonlinear Integer Programs with a Subgradient Approach on Parallel Computers

Robert Bixby
John Dennis
Zhijun Wu

CRPC-TR92022
June 1992

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Appeared in *SIAM News*, 25 (4), July, 1992.
Replaces CRPC-TR90019: "A Subgradient Algorithm for Solving Nonlinear Integer Programming Problems."

Solving Nonlinear Integer Programs with a Subgradient Approach on Parallel Computers

Robert Bixby, John Dennis, and Zhijun Wu

June 8, 1992

Many large, and hard, nonlinear integer programming problems—or more generally mixed-integer nonlinear programming problems—arise in both theoretical study and practical applications. No general and efficient solution to these problems can be found with traditional computers. Thus, it becomes necessary to develop new algorithms for solving these problems on advanced architectures, such as parallel computers. This article describes a new algorithm, developed at the Center for Research on Parallel Computation at Rice University, for solving nonlinear integer programs and shows how the solution to a nonlinear integer program can be achieved on a parallel computer.

Nonlinear integer programming is concerned with solving the problem

$$\begin{aligned} \textbf{min} \quad & f(x) \\ & x \in B^n = \{0,1\}^n \end{aligned} \tag{1}$$

or its natural extension

$$\begin{aligned} \textbf{min} \quad & f(x) \\ & x \in R^n \quad integral, \end{aligned} \tag{2}$$

where $f : R^n \longrightarrow R$ is a general nonlinear function.

This class of problems contains many NP-hard problems and has both theoretical and practical applications. For example, consider the problem that for any norm $\|\cdot\|$,

$$\begin{aligned} \textbf{min} \quad & \| \, b - Ax \, \| \\ & x \in R^n \qquad integral, \end{aligned} \tag{3}$$

where $b \in R^m$ and A is an $m \times n$ matrix with integer elements. This problem, called the closest vector problem in integer programming, has been proven to be NP-complete even for simple norms such as l_2 and l_∞.

Another example is related to the solution of a class of more general problems; mixed-integer nonlinear programming problems. It turns out that under some circumstances, problems of this class can be reduced to general nonlinear integer programs. For instance, an unconstrained mixed-integer nonlinear program,

$$\textbf{min} \quad g(x, y) \tag{4}$$

$$\begin{aligned} & y \in R^m \\ & x \in R^n \qquad \text{integral} \end{aligned} \tag{5}$$

can be formulated, under some appropriate assumptions, as the following problem:

$$\begin{aligned} \textbf{min} \quad & f(x) \\ & x \in R^n \qquad \text{integral}, \end{aligned} \tag{6}$$

with $f(x) = \textbf{min} \, \{g(x, y) : \; y \in R^n\}$.

Mixed-integer nonlinear programming has recently found an important application in the steady-state optimization of gas pipeline network operation. Percell [4] studied three model problems for pipeline networks with up to 12 compressor stations, each of which contains several compressor units. Given demands and resources for the network, the goal of the optimization is to find steady-state profiles of pressure, flow, temperature, and compressor station configurations (i.e., choice of compressor units to be run). These solutions are optimal for chosen objectives, such as minimizing the amount of fuel, maximizing the total flow, and maximizing gas inventory.

Mathematically, the problem is formulated as a minimization of a nonlinear objective function subject to nonlinear equality constraints. There are

two groups of variables: continuous and discrete. The continuous variables correspond to pressures, flows, temperatures, speeds, powers, etc. The discrete variables correspond to compressor stations, and their values are the number of compressor units to be run.

There are several well-studied methods for nonlinear continuous optimization. The challenge of applying them to the pipeline optimization problem lies in finding an effective way to handle the discrete variables, i.e., in deciding how many compressors should be used and which should be turned on or off for the network operation. Simply enumerating all possible values for the discrete variables is infeasible as there are exponentially many combinations with respect to the number of discrete variables. Besides, the number of discrete variables tends to be large in practice. Anglard and David [1] considered a gas pipeline network with 196 compressor stations, each of which contained up to eight compressor units. A robust way to deal with the discrete variables is to solve a nonlinear integer program, as illustrated in problem (6).

Several approaches to the solution of a nonlinear integer program have been studied in the last 30 years (see [3] for a general review). The main ones are enumeration, algebraic, and linearization approaches. Most of them work for problems with special structures. But for problems with general objective functions, such as problem (6) the enumeration method is hardly efficient, and the other two approaches cannot be applied owing to their special requirements for the forms of the objective function.

1 The Subgradient Approach

Bixby, Dennis, and Wu [2] have proposed a subgradient approach to nonlinear integer programming problems with more general or complicated objective functions. With this approach, a nonlinear integer program in the form of (1) is considered as a nonsmooth problem over the set of $0-1$ integer points. Notions of subgradients and supporting planes are then introduced for the objective function at integer points. By computing subgradients and supporting planes, a sequence of linear approximations to the objective function is constructed, and the optimal solution is found by successively solving the sequence of linear subproblems.

More specifically, the subgradient algorithm iteratively searches for the

solution among integer points. At each iteration, it generates the next iterative point by solving the problem for a local piecewise linear model that is constructed from the supporting planes for the objective function at the set of iterative points already generated. The supporting planes are computed by using special continuous optimization techniques. The problem for the local piecewise linear model in each iteration is equivalent to a linear integer minimax problem, which can be solved with a standard method for linear integer programming.

In Algorithm–1, f^r, the restriction of f to B^n, where $B^n = \{0,1\}^n$, is called the discrete objective function and $\partial f^r(x^{(i)})$ is the subdifferential of f^r at $x^{(i)}$. Formally, the algorithm can be outlined as follows:

Algorithm–1 {*The Subgradient Algorithm*}

0 {*Initialization*}
$T = \emptyset$, $H = \emptyset$, $i = 0$
pick up $x^{(i)} \in B^n$
1 {*Iteration*}
do while $i \leq m$
1.1 {*Optimality testing*}
if $x^{(i)} \in T$ or $0 \in \partial f^r(x^{(i)})$ is known
then
$x^{(i)}$ is the optimal solution, stop
1.2 {*Generating supporting planes*}
$T = T \cup \{x^{(i)}\}$
$H = H \cup \{g_{x^{(i)}} : g_{x^{(i)}}(x) = f^r(x^{(i)}) + s^T_{x^{(i)}}(x - x^{(i)}), \quad s_{x^{(i)}} \in \partial f^r(x^{(i)})\}$
1.3 {*Solving a linear integer minimax problem*}
find a solution $x^{(*)}$ for
$\mathbf{min}_{x \in B^n}$ {$p(x)$=**max** $\{g(x) : \ g \in H\}$}
1.4 {*Updating*}
$i = i + 1$
$x^{(i)} = x^{(*)}$
end do

There are three major steps at each iteration: optimality testing, generation of a supporting plane, and solution of a linear integer minimax problem.

The essential work in the first step is to construct the optimality criteria. The challenge of the second step lies in finding a method for computing a subgradient such that a supporting plane can be generated. For the third step a special linear integer program needs to be solved.

The optimality criteria are based on the following facts (given in [2]):

Fact-1: A **necessary** and **sufficient** condition for $x^* \in B^n$ to be the minimizer of f^r (or f) over B^n is $0 \in \partial f^r(x^*)$.

Fact-2: For the sequence $T = \{x^{(j)} \in B^n,\ j = 1,\ \ldots,\ i\}$ generated by the algorithm at the ith iteration, if $\exists j < i$ such that $x^{(j)} = x^{(i)}$, then $x^{(i)}$ must be an optimal solution.

It follows from **Fact-2** that the algorithm stops whenever an iterate is repeated. Because there are only finitely many distinct iterates, the algorithm is finite.

Consider the generation of a supporting plane for the function f^r at a given integer point $\bar{x}$. If g is the function for the supporting plane, then g is a linear function and

$$g(x) = f^r(\bar{x}) + s^T(x - \bar{x}) \qquad\qquad s \in \partial f^r(\bar{x}). \tag{7}$$

To obtain this function, $f^r(\bar{x})$ can be computed easily, but the subgradient s must be chosen such that g bounds f^r from below as tightly as possible. In the case where f is convex and differentiable, it is easy to verify that $\nabla f(\bar{x})$, the gradient of f at $\bar{x}$, is a subgradient of f^r at $\bar{x}$. A trivial way to choose s, therefore, is to set s to $\nabla f(\bar{x})$. However, with this subgradient, g could be too "steep" to be a preferred supporting plane; in this case a subgradient other than $\nabla f(\bar{x})$ is required, such that g is as "flat" or "close" to f^r as possible.

Unfortunately, there are no simple methods for computing any subgradients for general nonlinear nonsmooth functions. In this algorithm subgradients are obtained by a process that can successively improve a given subgradient. The process starts with the subgradient $s = \nabla f(\bar{x})$ and then updates it such that the corresponding supporting plane g is "lifted," i.e., made "flatter" or "closer" to f^r. The updated s remains a subgradient as long as g still supports f^r at $\bar{x}$, i.e.,

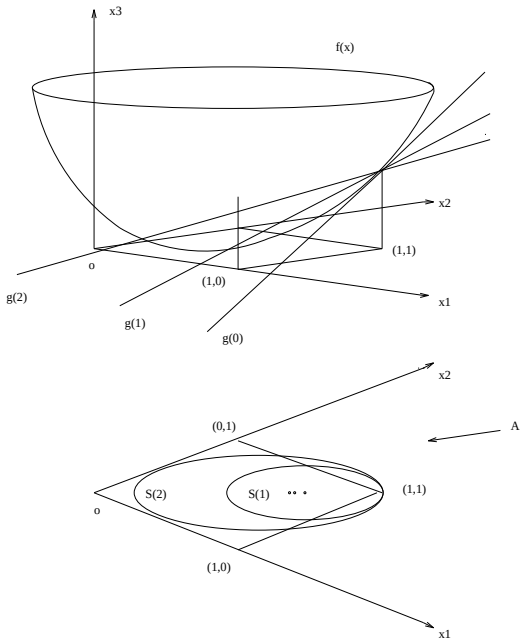

Figure 1: The lifting process for computing subgradients.

$$g(x) \leq f^r(x) \qquad \forall x \in B^n. \tag{8}$$

The lifting process continues until the best possible supporting plane is obtained. However, for every update, condition (8) must be verified. For a given subgradient s, if S is defined such that $x \in S$ if $f(x) \leq g(x)$, then condition (8) is equivalent to the following statement:

> A vector s is a subgradient of f^r if and only if the interior of S does not contain $0-1$ integer points.

Figure 1 illustrates with a simple example how the lifting process is conducted and condition (8) is guaranteed. In this example, the lifting process is applied to find a subgradient of f^r at $\bar{x}$. First, s is set to $\nabla f(\bar{x})$. The supporting plane defined by this subgradient is $g_{(0)}$. Then s is updated to "lift" $g_{(0)}$ a little bit, and $g_{(1)}$ and $S_{(1)}$ are obtained. Let $A = \{x \in R^n : x_i \geq 0, \text{ if } \bar{x}_i = 1, \text{ and } x_i \leq 1 \text{ if } \bar{x}_i = 0, \ i = 1, \ldots, n\}$. Geometrically, A

is a region that contains B^n, and its boundaries are formed by hyperplanes $x_i = 1 - \bar{x}_i,\ i = 1, \ldots, n$. Once it has been observed that the interior of A contains no points in B^n other than $\bar{x}$, condition (8) holds if $S_{(1)}$ is inside of A. In general,

> for each S obtained in the lifting process, the interior of S does not contain $0-1$ integer points as long as S is contained in A.

To obtain better subgradients, s can be further updated until the corresponding S hits the boundary of A (see $g_{(2)}$ and $S_{(2)}$ in Figure 1).

Now consider the updated subgradient s and its corresponding S. Let d_i be the distance between S and the ith boundary of A. Then $d = (d_1, \ldots, d_n)$ is a function of s. It can be proven that the function is well defined under some assumptions. The lifting process can then be formulated mathematically as an optimization problem:

$$\begin{array}{ll} \textbf{min} & \| \, d(s) \, \| \\ \textbf{st.} & d_i(s) \geq 0 \qquad i = 1, \ldots, n \end{array} \tag{9}$$

The major computation for this optimization problem is the evaluation of the function $d(s)$ for each s. In terms of the lifting process, distances between the boundaries of S and A need to be calculated for each lifting step. If the distances are positive, S is inside A and hence condition (8) holds. In any case, $d_i(s)$ for any i can be calculated by first computing an extreme point of S along the x_i direction and then calculating the distance between the extreme point and the ith boundary of A. The extreme point of S can be found by solving a relatively simple continuous optimization problem (linear objective function with only one nonlinear constraint).

Finally, the third step of each iteration involves the solution of a linear integer minimax problem. Because the problem can be formulated as a special linear integer program, a branch-and-bound procedure can be applied. To compute the bound for every branching step, the standard simplex method is used to solve the dual problem of the linear relaxation problem.

2 Generating a Supporting Plane in Parallel

As described in the preceding section, the subgradient algorithm requires that a number of subproblems be solved at every iteration. These subproblems are

difficult, and their solutions involve large amounts of computation. Reducing the time required to solve these subproblems appears to be a very important and challenging, consideration. Bixby, Dennis, and Wu [2] propose the use of parallel computers to speed up the algorithm so that reasonably large problems can be solved. In fact, parallelism can indeed be exploited for the algorithm to achieve high performance.

The subgradient algorithm carries at the top level an iterative procedure that is sequential and cannot be done in parallel. Numerical experiments show that for most problems the algorithm can find an optimal solution in only a few, at most $\mathcal{O}(n)$, iterations. Therefore, the algorithm can be effectively parallelized if the computation at each iteration can be done in parallel.

For each iteration the major computational costs are those of generating a supporting plane and solving a linear integer minimax problem. To generate the supporting plane, the lifting process is conducted to achieve a solution to problem (9), where most of the work is in the evaluation of the function $d(s)$, as discussed previously. Computing each component of $d(s)$ involves the solution of a continuous optimization subproblem. A total of n subproblems, therefore, need to be solved to obtain all components of $d(s)$ for each given s. The computation here could be very expensive.

Each of the n subproblems is totally independent, however, and they all have almost the same structure and size. Thus, it is easy to introduce parallelism to do the function evaluation, and if up to n processors are used, subproblems can be distributed evenly over the processors and solved in parallel with little communication overhead. Parallelism of this type is suitable for such parallel systems as the Intel iPSC/860 hypercube, with up to 128 processors, and the nCube, with up to 8192 processors. For large problems ($n = 100 \sim 1000$) very high performance can be achieved as many, up to n, processors can be used.

3 Parallel Branch-and-Bound for the Linear Subproblem

The linear integer minimax problem induced at each iteration by the subgradient algorithm is solved by applying a branch-and-bound procedure, a

popular scheme for solving linear integer programming problems. But for large problems—those with, say, 100 to 1000 variables—the method may still produce so many subproblems that the solution cannot be obtained in a reasonable time.

The branch-and-bound procedure can be represented by a tree, with nodes corresponding to subproblems and branches corresponding to relations among subproblems. The process can thus be parallelized by exploiting the tree structure, although it is not straightforward to do so. The tree structure is constructed dynamically as the procedure moves forward, and the parallelism among subproblems often is not known until the subproblems are generated.

Algorithm–2 is the parallel branch-and-bound procedure used within the subgradient algorithm for the linear subproblems. The algorithm is based on the general branch-and-bound method, but a multiple branching strategy is used instead of the more common binary branching. More precisely, if p processors are used, the algorithm always makes p branches at every branching step, producing p subproblems and solving them, one for each processor, in parallel. After solving the subproblems, the algorithm proceeds by making branches recursively for the new subproblems.

With the multiple branching strategy, processors can be scheduled in a systematic way: Each time p subproblems are produced, they are assigned to the p processors, one for each processor. All groups of p subproblems produced in this way are almost the same, except for some variables set to different values. The load is thus balanced automatically in solving the subproblems. Moreover, because subproblems are generated regularly and correspond to processors, first subproblem for first processor, second subproblem for second processor, etc., processors do not need to trace a global subproblem stack to find the subproblems they need to solve. Instead, each processor has only a small local stack of its own subproblems.

Globally, the parallel branch-and-bound procedure conducts a depth-first search because at each step the new subproblems always are processed first. But after every group of p subproblems is solved, they can be sorted according to some priority. The branching can then be made for the subproblems in the sorted order (local best-first branch).

Algorithm–2 {*The Parallel Branch-and-Bound Algorithm*}

```
* {Initial Procedure}
  initialize p, z_p, z and x (p represents the initial subproblem)
  solve p
  let z_p and x_p be the optimal value and solution
  if x_p is integral then
     z = min{z, z_p}, x = x_p, stop
  push(p, P) (P is a local subproblem stack)
  branch–and–bound(1, x, z)
  pop(P)
* {End of Initial Procedure}

* {Recursive Procedure}
  branch–and–bound(i, x, z)
    broadcast z_p from processor i
    if z_p ≥ z, return
    if processor # = i then
       select branching variables
    broadcast branching variables from processor i
    generate and solve subproblem p
    let z_p and x_p be the optimal value and solution
    if x_p is integral then
       z = min{z, z_p}, x = x_p
    update z and x if necessary
    push(p, P)
    for j = 1, ..., # of processors do
        branch–and–bound(j, x, z)
    pop(P)
* {End of Recursive Procedure}
```

4 Remarks

Computational experiments have been conducted with a parallel implementation of the subgradient algorithm on a 512-node nCube located at the California Institute of Technology. The program is written in Express C, an extended C language for distributed-memory parallel computers. In addition to standard C, the language provides a variety of message-passing functions.

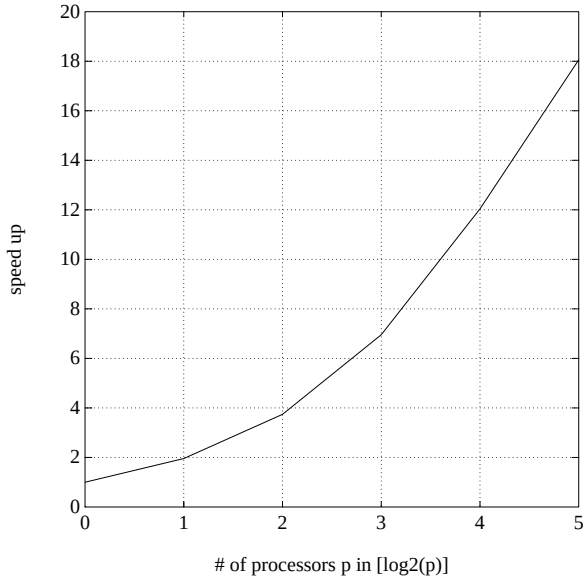

Figure 2: Speed–up obtained with varying numbers of processors.

Small problems (up to 64 dimensions) have been tested. Preliminary results show that most of the test problems can be solved by the subgradient algorithm in only a few, at most $\mathcal{O}(n)$, iterations, if proper initial guesses are used. For a test problem of dimension n there are 2^n $0-1$ integer points. In the worst case, therefore, the algorithm might need to run 2^n iterations, which cannot be done in a reasonable time, even for an average n, say 32, for which there are $4,294,967,296$ $0-1$ integer points in total. In reality, however, most of the test problems can be solved more efficiently than by exhausting all possible integer points. For test problems of dimension 32 or 64, with some initial guesses, only several iterations were taken.

For the parallel implementation of the program on the nCube, if the number of processors p ($p \leq n$) is doubled, the total computation time can often be reduced by almost half. Figure 2 shows the speedup that can be obtained for a 32-dimension test problem when different numbers of processors are used. The greatest speedup is about 18, which can be improved by testing larger problems.

References

[1] P. Anglard and P. David, *Hierarchical Steady–State Optimization of Very Large Gas Pipelines*, Pipeline Simulation Interest Group Annual Meeting, Toronto, Ontario, Canada (1988).

[2] R. Bixby, J. Dennis and Z. Wu, *A Subgradient Algorithm for Nonlinear Integer Programming and Its Parallel Implementation*, Technical Report, Center for Research on Parallel Computation, Rice University, Houston, TX, (1991).

[3] P. Hansen, B. Jaumard, V. Mathon, *Constrained Nonlinear 0–1 Programming*, RRR #47–89, RUTCOR, Rutgers University, New Brunswick, NJ, (1989).

[4] P.B. Percell, *Steady–State Optimization of Gas Pipeline Network Operation*, Pipeline Simulation Interest Group Annual Meeting, Tulsa, OK, (1987).

John Dennis and Robert Bixby are professors in the Department of Mathematical Sciences at Rice University. Zhijun Wu is a postdoctoral research associate at the Advanced Computing Research Institute at Cornell University.