RC25103 (W1101-112) January 28, 2011
Computer Science

IBM Research Report

A Probing Algorithm for MINLP with Failure Prediction by SVM

Giacomo Nannicini[1], Pietro Belotti[2], Jon Lee[3], Jeff Linderoth[4], François Margot[1], Andreas Wächter[3]

[1]Tepper School of Business
Carnegie Mellon University
Pittsburgh, PA

[2]Department of Mathematical Sciences
Clemson University
Clemson, SC

[3]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

[4]Industrial and Systems Engineering
University of Wisconsin-Madison
Madison, WI

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

A probing algorithm for MINLP with failure prediction by SVM

Giacomo Nannicini[1⋆], Pietro Belotti[2], Jon Lee[3], Jeff Linderoth[4⋆⋆], François Margot[1⋆⋆⋆], Andreas Wächter[3]

[1] *Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA*
{nannicin,fmargot}@andrew.cmu.edu
[2] *Dept. of Mathematical Sciences, Clemson University, Clemson, SC*
pbelott@clemson.edu
[3] *IBM T. J. Watson Research Center, Yorktown Heights, NY*
{jonlee,andreasw}@us.ibm.com
[4] *Industrial and Systems Eng., University of Wisconsin-Madison, Madison, WI*
linderoth@wisc.edu

Abstract. Bound tightening is an important component of algorithms for solving nonconvex Mixed Integer Nonlinear Programs. A *probing* algorithm is a bound-tightening procedure that explores the consequences of restricting a variable to a subinterval with the goal of tightening its bounds. We propose a variant of probing where exploration is based on iteratively applying a truncated Branch-and-Bound algorithm. As this approach is computationally expensive, we use a Support-Vector-Machine classifier to infer whether or not the probing algorithm should be used. Computational experiments demonstrate that the use of this classifier saves a substantial amount of CPU time at the cost of a marginally weaker bound tightening.

1 Introduction

A Mixed Integer Nonlinear Program (MINLP) is a mathematical program with continuous nonlinear objective and constraints, where some of the variables are required to take integer values. Without loss of generality, we assume that the problem is a minimization problem. MINLPs naturally arise in numerous applied problems, see e.g. [1, 2]. In this paper, we address nonconvex MINLPs where neither the objective function nor the constraints are required to be convex — a class of problems typically difficult to solve in practice. An exact solution method for nonconvex MINLPs is Branch-and-Bound [3], where lower bounds are obtained by convexifying the feasible region using under-estimators, often linear inequalities [4, 5]. The convexification depends on the variable bounounds,

⋆ Supported by an IBM grant and by NSF grant OCI-0750826.
⋆⋆ Supported by U.S. Department of Energy grant DE-FG02-08ER25861 and by NSF grant CCF-0830153.
⋆⋆⋆ Supported by NSF grant OCI-0750826.

with tighter bounds resulting generally in a tighter convexification. As such, bound tightening is an important part of any MINLP solver.

Probing is a bound-tightening technique often applied to Mixed Integer Linear Programs (MILPs) [6]. The idea is to tentatively fix a binary variable to 0 and then to 1, and use the information obtained to strengthen the linear relaxation of the problem. Similar techniques have been applied to MINLPs as well [5]. In this paper, we propose a probing technique based on truncated Branch-and-Bound searches. Let $\bar{z}$ be the objective value of the best solution of the original problem found so far. In each Branch-and-Bound search, we choose a variable, say x_i, and impose $x_i \in S$, where S is a subinterval of the current domain of x_i. In addition, we add a constraint bounding the objective value of the solution to at most $\bar{z}$. If that problem is infeasible, we can discard S from the domain of x_i. On the other hand, if we are able to solve the modified problem to optimality, with an optimal value $\bar{z}^* < \bar{z}$, we update $\bar{z}$ and can again discard S from the domain of x_i. Details on the choice of x_i and S are given in Section 3.

This probing algorithm potentially requires a significant amount of CPU time. To limit this drawback, we use a Support Vector Machine (SVM) classifier [7] before performing a Branch-and-Bound search, to predict the success or failure of the search. If we conclude that the probing algorithm is unlikely to tighten the bounds on the variable, we skip its application. Machine learning methods have been used in the OR community for various tasks, such as parameter tuning [8] and solver selection [9]. In this paper, machine learning is used to predict failures of an algorithm based on characteristics of its input data. The features on which the SVM prediction is based are problem and subinterval dependent, and are related to the outcome of the application of a fast bound-tightening technique (Feasibility-Based Bound Tightening [5]) using the same subinterval.

We provide preliminary computational results to assess the practical efficiency of the approach. The experiments show that the proposed probing algorithm is very effective in tightening the variable bounds, and it is helpful for solving MINLPs with Branch-and-Bound. By using SVM to predict failures of the probing algorithm, we save on average 30% of the total bound-tightening time, without much deterioration of the quality of the bounds.

The rest of this paper is organized as follows. In Section 2, we introduce the necessary background. In Section 3, we describe the probing algorithm. In Section 4, we discuss how we can integrate a machine learning method in our algorithm to save CPU time. In Section 5, we provide computational testing of the proposed ideas and Section 6 has conclusions.

2 Background

A function is *factorable* if it can be computed in a finite number of simple steps, starting with model variables and real constants, using elementary unary and

binary operators. We consider an MINLP of the form:

$$\left.\begin{array}{rl} \min & f(x) \\ s.t. & g_j(x) \leq 0 \quad \forall j \in M \\ & x_i^L \leq x_i \leq x_i^U \quad \forall i \in N \\ & f(x) \leq \bar{z} \\ & x_i \in \mathbb{Z} \quad \forall i \in N_I, \end{array}\right\} \qquad \mathcal{P}$$

where f and g_j are factorable functions, $N = \{1, \dots, n\}$ is the set of variable indices, $M = \{1, \dots, m\}$ is the set of constraint indices, $x \in \mathbb{R}^n$ is the vector of variables with lower/upper bounds $x^L \in (\mathbb{R} \cup \{-\infty\})^n$, $x^U \in (\mathbb{R} \cup \{+\infty\})^n$, and $\bar{z}$ is an upper bound on the optimal objective value, which can be infinite. The variables with indices in $N_I \subset N$ are constrained to take on integer values in the solution.

A Linear Programming (LP) based Branch-and-Bound algorithm can be used to solve $\mathcal{P}$ [4]. In such a method, subproblems of $\mathcal{P}$ are generated by restricting the variables to reduced interval domains, $[\bar{x}^L, \bar{x}^U] \subset [x^L, x^U]$. A key step is the creation of an LP relaxation of the feasible region of a subproblem, which we refer to as *convexification*. This convexification is used to obtain a lower bound on the optimal objective value of the subproblem. In general, the tighter the variable bounds, the tighter the convexification, and the stronger the resulting lower bound. Therefore, bound-tightening techniques aim to deduce improved variable bounds implied by the constraint structure of the subproblem, and are widely used by existing software, such as `Baron` [10] and `Couenne` [11], for the solution of MINLPs.

A commonly used bound-tightening procedure is Feasibility-Based Bound Tightening (FBBT), which uses a symbolic representation of the problem in order to propagate bound changes on a variable to other variables. For instance, suppose that $\mathcal{P}$ contains the equation $x_3 = x_1 + x_2$, with variable bounds $x_1 \in [0, 1]$, $x_2 \in [0, 3]$, $x_3 \in [0, 4]$; if we tighten the bounds on x_2 and restrict this variable to the interval $[1, 2]$, then we can propagate the change to x_3 and impose $x_3 \in [1, 3]$. A full description of FBBT can be found in [12, 13].

The other aspects of the Branch-and-Bound algorithm are similar to those of any Branch-and-Bound for solving MILPs; see [5] for more details.

3 The probing algorithm

In this section we describe the probing algorithm to increase the lower bound on variable x_i, where the current bounds on that variable are $x_i \in [x_i^L, x_i^U]$ with $x_i^L > -\infty$. The special case $x_i^L = -\infty$ is treated below. The probing algorithm for decreasing the upper bound is similar. For simplicity, we describe the procedure applied to the root node $\mathcal{P}$.

Let ℓ and u be such that $x_i^L \leq \ell \leq u \leq x_i^U$. We denote by $\mathcal{P}_i[\ell, u]$ the problem obtained from $\mathcal{P}$ by adding the constraint $x_i \in [\ell, u]$. For $s > 0$, an *s-probing iteration* for x_i consists of the following: set $\ell = x_i^L$, $u = \min\{x_i^L + s, x_i^U\}$, and perform a Branch-and-Bound search on $\mathcal{P}_i[\ell, u]$ *with a time limit*. If we have then

proved that $\mathcal{P}_i[\ell, u]$ is infeasible, we update the lower bound $x_i^L \leftarrow u$. If we are able to solve $\mathcal{P}_i[\ell, u]$ to optimality, finding a solution with objective value z^*, we update the best incumbent value $\bar{z} \leftarrow z^*$ and the lower bound $x_i^L \leftarrow u$. In both cases, the s-probing iteration is deemed a success. Otherwise, it is a failure.

The Aggressive Probing algorithm for variable x_i (see Algorithm 1) has an initial value for s as input and runs an s-probing iteration. While an exit condition is not met, if the s-probing iteration is successful, the value of s is doubled and a new s-probing iteration is executed. If an s-probing iteration fails, the value of s is halved and a new s-probing iteration is performed.

For integer variables, we round the probing interval endpoints appropriately. Additionally, if the search on $P_i[\ell, u]$ is completed and u is integer, we set $x_i^L \leftarrow u + 1$ instead of $x_i^L \leftarrow u$.

With sufficiently large time limits, Algorithm 1 will provide an optimality certificate if $\bar{z}$ is the optimal objective value. If run to completion, the algorithm proves that no better solution exists with x_i in the interval $[x_i^L, x_i^U]$.

The special case $x_i^L = -\infty$ is handled as follows. Define a positive number B. We perform a Branch-and-Bound search on $\mathcal{P}_i[-\infty, -B]$. If $\mathcal{P}_i[-\infty, -B]$ is proved infeasible or solved to optimality within the time limit, we set $x_i^L \leftarrow -B$ and execute Algorithm 1. Otherwise, we conclude that x_i^L cannot be tightened. In our experiments, we use $B = 10^{10}$.

Two details of the algorithm still need to be specified: the exit condition and the initial choice of s. We use two exit conditions: a maximum CPU time for the application of Aggressive Probing, and a maximum number of consecutive failed s-probing iterations. We experimented with several choices for the initial value of s that took into account the distance between the variable bounds and the solution of the LP relaxation (or a feasible solution to $\mathcal{P}$, if available). But the results were not better than a simpler method that seems to work well: the initial value of s is chosen to be a small, fixed value `size`, depending on the variable type. In our experiments, we use `size` $= 0.5$ for continuous variables, `size` $= 1.0$ for general integer variables, and `size` $= 0.0$ for binary variables.

The good performance of the update strategy and the initial choice for the value of s lies in the dynamic and geometric adjustment of the interval length during Aggressive Probing. If the initial interval can easily be proven infeasible, the s-probing iteration will terminate very quickly, typically with a single application of FBBT, or at the root node by solving the LP relaxation. In this case, because the interval size is increased in a geometric fashion, in a few iterations we will reach the scale that is needed for the probing interval to be "not trivially infeasible". On the other hand, using a small interval size yields better chances of completing the s-probing iteration within the time limit, in case $\mathcal{P}_i[\ell, u]$ is difficult to solve even with u close to ℓ.

In our experiments, we set the time limit for the Branch-and-Bound search during an s-probing iteration to $\min\{2/3\,\texttt{time_limit}, \texttt{time_limit} - \mathit{current_time}\}$. This avoids investing all of the CPU time in the first probing iteration in cases where the initial interval-size guess is too large.

Algorithm 1 The AGGRESSIVE PROBING algorithm.

INPUT: variable index i, `time_limit`, `size`, `max_failures`
Set $s \leftarrow$ `size`, $fail \leftarrow 0$,
while $current_time <$ `time_limit` **and** $fail <$ `max_failures` **and** $x_i^L < x_i^U$ **do**
  $\ell \leftarrow x_i^L$
  Set $u \leftarrow \min\{\ell + s, x_i^U\}$; if x_i is integer constrained, round $u \leftarrow \lfloor u \rfloor$
  Execute limited-time Branch-and-Bound on $\mathcal{P}_i[\ell, u]$
  if solution $\bar{x}$ found **then**
    Set $\bar{z} \leftarrow \min\{\bar{z}, f(\bar{x})\}$
  if search complete **then**
    if x_i is integer constrained **then**
      Set $x_i^L \leftarrow u + 1$
    else
      Set $x_i^L \leftarrow u$
    Set $s \leftarrow 2s$, $fail \leftarrow 0$
  else
    Set $s \leftarrow s/2$, $fail \leftarrow fail + 1$

4 Support Vector Machine for failure prediction

Applying AGGRESSIVE PROBING to tighten all variables of even a moderately-sized MINLP can take a considerable amount of CPU time. We would like to avoid wasting time in trying to tighten the bounds of a variable for which there is little hope of success.

Observe that if a probing subproblem $\mathcal{P}_i[\ell, u]$ is not solved to optimality, the CPU time invested in that probing iteration is wasted. To avoid this situation, we suggest to use the degree of success in applying the fast FBBT algorithm on $\mathcal{P}_i[\ell, u]$ as a factor in deciding whether or not to run the expensive AGGRESSIVE PROBING algorithm. Note that FBBT is the first step of the Branch-and-Bound algorithm used in AGGRESSIVE PROBING, so this does not require additional work. Our hypothesis is that if the constraint $x_i \in [\ell, u]$ used during an s-probing iteration does not result in tighter bounds on other variables when FBBT is used, then $\mathcal{P}_i[\ell, u]$ is approximately as difficult as $\mathcal{P}$, so the limited-time Branch-and-Bound algorithm is likely to fail. This intuition is confirmed by empirical tests: on 84 nontrivial MINLP instances taken from various sources, we perform AGGRESSIVE PROBING with `max_failures` = 10 to tighten lower and upper bound of all variables, processing them in the order in which they appear in the problem, with a time limit of 1 minute per variable and a total time limit of 1 hour per instance. We record, for each s-probing iteration, whether FBBT is able to use the probing interval $x_i \in [\ell, u]$ to tighten the bounds on other variables. We observe that, in 587 cases out of 11,747, no stronger bounds are obtained. In 528 of these 587 cases (90%), the subsequent Branch-and-Bound search on $\mathcal{P}_i[\ell, u]$ could not be completed within the time limit. Therefore, the success of FBBT in using $x_i \in [\ell, u]$ to tighten other variable bounds indeed gives an indication of the difficulty of solving $\mathcal{P}_i[\ell, u]$.

Supported by this observation, we use the following strategy. Before applying Branch-and-Bound to $\mathcal{P}_i[\ell, u]$, we execute FBBT and compute a measure of the bound reduction obtained on the variables x_j, $j \in N \setminus \{i\}$. Based on this measure, we use an algorithm to decide whether to perform Branch-and-Bound on $\mathcal{P}_i[\ell, u]$. In the remainder of this section we discuss our choice of the bound reduction measure and the decision method.

4.1 Measuring the effect of FBBT

Several bound-reduction measures are possible. Because our aim is to save CPU time, the bound-reduction measure computation should be fast. A simple way of measuring the bound reduction obtained for the variables x_j, $j \in N \setminus \{i\}$, is to count the number of tightened variables, and for each of these, to compute the interval reduction: $\gamma(x_j) = 1 - (\tilde{x}_j^U - \tilde{x}_j^L)/(x_j^U - x_j^L)$, where $\tilde{x}^L$ (resp. $\tilde{x}^U$) are the vectors of variable lower (resp. upper) bounds after applying FBBT, and x^L, x^U are those of the original problem $\mathcal{P}$. (Infinity is treated like a number in the following way: $1/\infty = 0, \infty/2\infty = 0.5$.) Hence, to quantify the bound reduction associated with the application of FBBT on the problem $\mathcal{P}_i[\ell, u]$, we use a vector $(\eta, \rho) \in [0, 1]^2$, where η is the fraction of tightened variables, and ρ is the average value of $\gamma(x_j)$ over all x_j that were successfully tightened.

4.2 Support Vector Machines

Once a vector (η, ρ) is computed for variable x_i, the decision of whether to perform Branch-and-Bound is taken by a predictor trained by a Support Vector Machine (SVM). While it could be argued that SVM is not really required for classifying our 2-dimensional data, we use SVM for three reasons. First, in our experiments SVM performs better than a predictor based on a simple Gaussian model for the data, see Section 5.2. Second, our future research efforts will utilize additional input features besides (η, ρ) (see Section 6 for details); therefore, the flexibility and extensibility of SVM is desirable. Finally, SVM is a parameterized method allowing better control of the trade-off between Precision and Recall of the classifier (see below for details) than simpler methods. Because we are interested in a classifier with high Precision and good Recall, the ability to tune the classifier is an advantage.

Next, we provide a brief description of the basic concepts behind SVM; see [14, 15] for a comprehensive introduction to the topic. Given training data $\mathcal{D} = \{(z_i, y_i) : z_i \in \mathbb{R}^p, y_i \in \{-1, 1\}, i \in 1, \dots, q\}$, SVM seeks an affine hyperplane that separates the points with label -1 from those with label 1 by the largest amount, i.e., the width of the strip containing the hyperplane and no data point in its interior is as large as possible.

In its simplest form, the associated optimization problem can be written as:

$$\left.\begin{array}{lll} \min & \|h\|_2 & \\ s.t. & y_i(h^\top z_i - b) \geq 1 & \forall (z_i, y_i) \in \mathcal{D} \\ & h \in \mathbb{R}^p, b \in \mathbb{R}, & \end{array}\right\} \qquad \text{(SVM)}$$

where the hyperplane is defined by $h^\top z = b$. Instead of seeking a separating hyperplane in $\mathbb{R}^p$, which may not exist, SVM implicitly maps each data point into a higher dimensional *feature space* where linear separation may be possible. The mapping is implicit because we do not need explicit knowledge of the feature space. In the optimization problem (SVM), we express the separating hyperplane in terms of the training points z_i (see e.g., [15]), and substitute the dot-products between vectors in $\mathbb{R}^p$ with a possibly nonlinear *kernel function* $K : \mathbb{R}^p \times \mathbb{R}^p \mapsto \mathbb{R}$. The kernel function can be interpreted as the dot-product in the higher-dimensional space. The separation hyperplane in the feature space translates into a nonlinear separation surface in the original space $\mathbb{R}^p$. Furthermore, SVM handles data that is not separable in the feature space by using a *soft margin*, i.e., allowing the optimal separation hyperplane to misclassify some points, imposing a penalty for each misclassification. The outcome of the SVM training algorithm is a subset V of $\{z_i : \exists y \in \{-1, 1\} \text{ with } (z_i, y) \in \mathcal{D}\}$ with corresponding scalar multipliers $\alpha_v : v \in V$, and a scalar b. The elements of V are called *support vectors*. To classify a new data point $w \in \mathbb{R}^p$, we compute the value of $\sum_{v \in V} \alpha_v K(v, w) - b$ and use its sign to classify w. Hence, the complexity of storing an SVM model depends on the number of support vectors $|V|$, and the time required to classify a new data point depends on $|V|$ and K.

Commonly used kernel functions are:

- linear: $K(u, v) = u^\top v$,
- polynomial: $K(u, v) = (\lambda u^\top v + \beta)^d$,
- radial basis: $K(u, v) = e^{-\lambda \|u - v\|^2}$,

where λ, β and d are input parameters. Problem-specific kernel functions can be devised as well. Another commonly adjusted tuning parameter is the misclassification cost ω, which determines the ratio between the penalty paid for misclassifying an example of label 1 and the penalty paid for misclassifying an example of label -1. The ratio ω can be adjusted to handle unbalanced data sets where one class is much more frequent than the other.

4.3 Aggressive probing failure prediction with SVM

In this section we assume that we have an SVM model trained on a data set of the form $\mathcal{D} = \{(\eta_i, \rho_i, y_i) : (\eta_i, \rho_i) \in [0, 1]^2, y_i \in \{-1, 1\}, i = 1, \ldots, q\}$, where each point corresponds to an s-probing iteration, η_i, ρ_i are as defined in Section 4.1, and $y_i = 1$ if the limited-time Branch-and-Bound search applied to the corresponding probing subproblem did not complete, $y_i = -1$ otherwise. Generating the set $\mathcal{D}$ and computational experiments with model training will be discussed in Section 5.

Given such an SVM model, we proceed as follows. At an s-probing iteration corresponding to problem $\mathcal{P}_i[\ell, u]$, we apply FBBT and compute the resulting $w_j = (\eta_j, \rho_j)$ as described in Section 4.1. If $w_j = (0, 0)$, FBBT could not tighten the bounds on any variable; in this case, as discussed at the beginning of Section 4, we do not execute Branch-and-Bound and continue to the subsequent

probing iteration as if the s-probing iteration failed. If $w_j \neq (0,0)$, we predict the label y_j of w_j using our SVM classifier. The Branch-and-Bound search on $\mathcal{P}_i[\ell, u]$ is thus executed only if the predicted label is $y_j = -1$; otherwise, we continue with the algorithm as if the s-probing iteration failed.

Note that we could apply the SVM classifier even on points of the form $(0,0)$. However, in our experiments this point was always labeled as 1 by the tested SVM models, therefore we save CPU time by not running the SVM predictor. Additionally, we exclude points $(0,0,y_i)$ from the data set $\mathcal{D}$ on which the model is trained; this yields an additional advantage that will be discussed in Section 5.

5 Computational experiments

We implemented Aggressive Probing within `Couenne`, an open-source Branch-and-Bound solver for nonconvex MINLPs [11]. We are mainly interested in applying our probing technique to difficult instances $\mathcal{P}$ to improve a Branch-and-Bound search; thus, in our implementation Aggressive Probing reuses as much previously computed information as possible. The root node of each probing subproblem $\mathcal{P}_i[\ell, u]$ is generated by modifying the root node of $\mathcal{P}$, changing variable bounds and possibly generating new linear inequalities to improve the convexification, so that the problem instance is read and processed only once. The branching strategy of `Couenne` was set to `Strong Branching` [5, 16] in all experiments.

We utilized LIBSVM [17], a library for Support Vector Machines written in C. Given the availability of LIBSVM's source code, it could be efficiently integrated within `Couenne` for our tests. The experiments were conducted on a 2.6 GHz AMD Opteron 852 machine with 64GB of RAM, running Linux.

5.1 Test instances

The test instances are a subset of MINLPLib [18], a freely available collection of convex and nonconvex MINLPs. We excluded instances with more than 1,000 variables and instances for which the default version of `Couenne` took more than 2 minutes to process the root node, or ran into numerical problems. Additionally, we excluded the instances for which Aggressive Probing was able to find the optimal solution and provide an optimality certificate in less than 2 hours. These are easy instances that can be quickly solved by default `Couenne`, therefore there is no need for expensive bound-tightening methods. We are left with 32 instances, which are listed in Table 1.

5.2 Training the SVM classifier

As a first step in training an SVM to classify failures of the probing algorithm, we obtained a large-enough set of training examples. We used a superset of the test problems described in Section 5.1, including some additional problems from MINLPLib as well as problems from [19] with less than 1,000 variables, giving a

			Without SVM						With SVM					
	# vars		Probing		Probing + FBBT + Conv.				Probing		Probing + FBBT + Conv.			
Instance	orig	total	Tght. %	Red. %	Tght. %	Red. %	Gap %	Time	Tght. %	Red. %	Tght. %	Red. %	Gap %	Time
csched2	401	582	2.00	5.05	3.44	7.12	97.76	36474.7	1.25	4.74	1.20	17.50	97.76	288.2
fo7_2	115	253	12.17	25.00	13.04	43.62	0.00	11133.1	12.17	7.87	13.04	35.37	0.00	6731.2
fo7	115	253	10.43	31.84	9.88	57.82	0.00	11129.9	10.43	5.87	9.88	43.28	0.00	5487.6
fo8_ar2_1	145	371	46.90	50.12	38.81	47.23	0.00	16372.5	46.90	49.99	38.81	47.15	0.00	13494.5
fo8_ar25_1	145	371	46.90	50.97	38.81	47.70	0.00	16373.4	46.90	50.22	38.81	47.24	0.00	13523.9
fo8_ar3_1	145	371	51.72	48.53	46.63	41.42	0.00	16448.0	52.41	46.31	48.25	39.41	0.00	14474.2
fo8_ar5_1	145	371	49.66	47.91	42.59	43.50	0.00	16385.6	49.66	48.06	42.59	43.60	0.00	14715.3
fo8	147	325	9.52	24.80	8.92	55.21	0.00	14186.1	8.16	3.23	8.31	46.29	0.00	6107.0
fo9_ar2_1	181	462	47.51	49.45	39.18	46.19	0.00	20435.2	47.51	46.50	39.18	44.30	0.00	13768.2
fo9_ar25_1	181	462	47.51	49.57	39.18	46.16	0.00	20439.2	47.51	46.45	39.18	44.18	0.00	13821.3
fo9_ar3_1	181	462	49.72	50.20	42.42	44.66	0.00	20433.2	49.72	45.41	42.42	41.80	0.00	17838.2
fo9_ar4_1	181	462	49.72	48.85	42.42	43.70	0.00	20443.4	49.72	44.90	42.42	41.48	0.00	17977.8
fo9_ar5_1	181	462	49.72	48.26	42.42	43.29	0.00	20437.1	49.72	44.95	42.42	41.44	0.00	18037.8
fo9	183	406	8.20	16.81	7.88	52.66	0.00	17546.6	8.20	2.50	7.88	45.12	0.00	7021.7
lop97icx	987	1393	0.00	0.00	0.00	0.00	0.00	120602.0	0.00	0.00	0.00	0.00	0.00	21.1
nvs23	10	64	90.00	92.07	100.00	98.41	97.02	1080.8	90.00	91.80	100.00	98.38	97.10	1080.7
nvs24	11	76	90.91	88.25	100.00	97.47	94.96	1201.0	90.91	88.25	100.00	97.47	94.96	1201.2
o7_2	115	253	19.13	26.50	19.37	46.38	0.00	11292.3	19.13	26.50	19.37	46.38	0.00	10191.1
o7_ar4_1	113	290	54.87	47.15	48.28	47.16	0.00	12918.8	54.87	47.25	48.28	47.22	0.00	12918.4
o7	115	253	17.39	28.12	16.21	53.76	0.00	11294.3	17.39	27.97	16.21	53.67	0.00	6872.0
o8_ar4_1	145	371	59.31	45.35	52.29	47.52	0.00	16696.9	59.31	45.80	52.29	47.81	0.00	16697.2
o9_ar4_1	181	462	56.35	45.76	49.13	45.96	0.00	20717.9	56.35	45.76	49.13	45.96	0.00	20474.3
space25a	384	502	17.19	38.16	32.67	30.61	0.00	36746.0	16.93	36.64	29.28	32.22	0.00	36704.2
tln12	169	361	35.50	35.67	37.95	34.10	0.00	19243.3	35.50	35.67	37.95	34.10	0.00	17973.4
tln5	36	81	41.67	33.33	62.96	44.86	0.00	3997.8	41.67	33.33	62.96	44.86	0.00	3997.8
tln6	49	109	61.22	28.33	71.56	41.24	0.00	5532.1	61.22	28.33	71.56	41.24	0.00	5619.6
tln7	64	141	43.75	25.60	64.54	32.90	0.00	7285.5	43.75	25.60	64.54	32.90	0.00	7281.3
tls12	813	1285	14.39	67.42	32.14	48.11	0.00	129660.0	14.39	67.42	32.14	48.11	0.00	128592.0
tls6	216	359	2.78	100.00	25.07	45.60	0.00	18514.3	2.78	100.00	25.07	45.60	0.00	18401.0
water4	196	319	25.51	56.54	40.13	50.49	41.54	18305.9	27.55	60.80	41.69	53.47	50.47	18013.8
waterx	71	174	14.08	25.38	32.18	20.77	34.83	7926.7	15.49	23.08	34.48	19.36	34.83	7874.8
waterz	196	319	15.31	62.02	25.71	55.10	21.47	18637.7	14.80	67.28	24.14	60.07	29.35	18617.8
Avg.	197.41	388.28	23.92	43.53	30.50	45.65	12.11	22496.6	23.90	40.58	30.32	44.59	12.64	15494.3

Table 1. Performance of Aggressive Probing, with and without failure prediction by SVM. We report, after probing (columns "Probing") and after applying FBBT and recomputing the convexification (columns "Probing + FBBT + Conv."): the fraction of tightened variables with finite bounds, and the average bound reduction. We also report the amount of optimality gap closed by the new convexification, and the total probing time.

total of 84 instances. We applied Aggressive Probing on all variables, with a time limit of 30 seconds for each s-probing iteration, 1 minute for each variable bound, and 2 hours per problem instance. We did not include data for s-probing iterations started with less than 20 seconds of CPU time left within the time limit, or iterations in which a feasible MINLP solution was discovered during the Branch-and-Bound search. For the remaining s-probing iterations, we recorded the values of (η_i, ρ_i) (see Section 4.1) and a label $y_i = 1$ if the probing iteration fails, $y_i = -1$ if it succeeds. The reason for excluding s-probing iterations performed with less than 20 seconds of CPU time left is that they are likely to fail simply because they are not given enough time to complete, regardless of the difficulty of the s-probing subproblem. Similarly, we excluded s-probing iterations in which an improved solution was found, because such a discovery cannot be predicted by only considering the s-probing subproblem, yet it can be used to infer tighter variable bounds through FBBT, therefore making $\mathcal{P}_i[\ell, u]$ easier than initially estimated. This yields a data set $\mathcal{D}$ with $q = 11,747$ data points that can be used for training, as explained in Section 4.3. Eliminating all points with $(\eta_i, \rho_i) = (0, 0)$ (see end of Section 4.3) leaves $11,160$ points, of which $4,186$ have the label $y_i = 1$. By removing the points $(0, 0)$ from the training set, the number of support vectors in the final model is likely to be smaller.

It is known that SVM is very sensitive to its algorithm settings, hence a grid search on a set of input parameters is typically applied in order to find the values that yield the best performance on the input data. We tested three types of kernel functions: linear, polynomial, and radial. For each of these kernel functions, we performed grid search on the input parameters (see end of Section 4.2), using the following values: $\lambda = 2^k$ with $k = -3, \ldots, 2$, $\beta = 2^k$ with $k = -3, \ldots, 2$, $d = 1, \ldots, 5$, and $\omega = 2^k$ with $k = -3, \ldots, 4$. Each parameter was considered only when appropriate; e.g., d was used for the polynomial kernel only. Overall, we tested 1,057 combinations of input parameters.

In our first set of experiments pertaining to the training of an SVM on $\mathcal{D}$, we performed 3-fold cross validation. We trained the model on 2/3 of $\mathcal{D}$, and used the remaining 1/3 to estimate the performance of the model. The resulting model consisted of 4,500 to 5,000 support vectors. Such a large number of support vectors would yield a slow classifier and may indicate overfitting. To obtain a model with fewer support vectors, we first attempted ν-classification [14], without success. In the end, training the model on a small subset of the full data set $\mathcal{D}$ was found to be very effective in reducing the number of support vectors, without deterioration in the accuracy of the model; this approach has been used before in the machine learning community [20].

With this setup, experiments for testing parameter values of the model were performed as follows. We randomly selected 10 different training sets, each one containing 1/10 of the full data set $\mathcal{D}$. Each training experiment, corresponding to a set of input parameter values, was performed on each of the 10 training sets, and the performance of the resulting models was evaluated on the 9/10 of $\mathcal{D}$ that was not used for training. To measure performance, we use Precision and

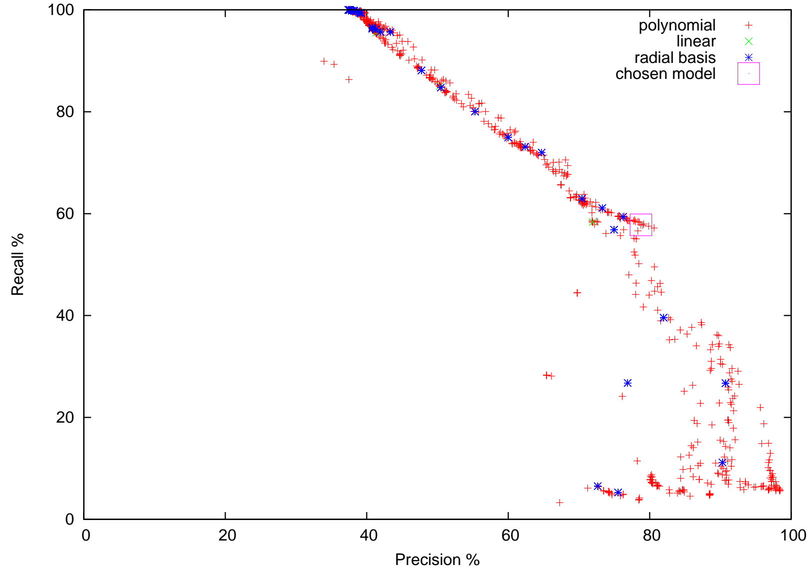

Fig. 1. Average values of Precision and Recall for all tested combinations of training input parameters.

Recall, commonly defined as follows:

$$\text{Precision: } TP/(TP + FP), \qquad \text{Recall: } TP/(TP + FN),$$

where TP is the number of True Positives, i.e., the examples with label 1 that are classified with label 1 by the model. FP is the number of False Positives, i.e., the examples with label -1 that are classified with label 1. Finally, FN is the number of False Negatives, i.e., the examples with label 1 that are classified with label -1. Intuitively, Precision is the fraction of data points labeled 1 by the classifier that are indeed of class 1, whereas Recall is the fraction of class 1 data points processed by the classifier that are correctly labeled as 1. Overall, for each set of training parameters, we have 10 values for Precision and 10 values for Recall. We compute the average and the standard deviation of these values, and use them to choose the best set of parameters.

Results of this experiment are summarized in Figure 1. Each point represents the average values of Precision and Recall corresponding to a set of parameter values. When producing the figure, we eliminated points for which the standard deviation of either Precision or Recall was more than 1/4 of its mean, because these points correspond to experiments with unreliable results.

Figure 1 shows the trade-off between Precision and Recall that can be achieved by varying the learning parameters. We are interested in the set of Pareto optima with respect to these two criteria. Most Pareto optima are obtained with a polynomial kernel, and the remaining with a radial-basis kernel. The linear

kernel yields inferior results, implying that the data set is difficult to separate in the original space. There are points with very high Precision ($> 85\%$) but low Recall that represent "conservative" classifiers: very few probing iterations are labeled as 1 (failure), but in that case the classifier is almost always correct. Such a classifier is of limited value for our purpose. We are more interested in the region with roughly 80% Precision and 60% Recall: approximately 60% of the unsuccessful probing iterations in the test set are predicted correctly, while keeping good Precision. These models use a polynomial kernel with degree $d \geq 3$ and $\omega = 1$; additionally, we found that using $\beta = 1, 2$, or 4 seems effective. These models have between 500 and 800 support vectors. The standard deviation of Precision and Recall for all models achieving a Pareto optimum is fairly small, typically less than 2. Therefore, we can assume that the performance of the SVM model does not depend heavily on the particular subset of $\mathcal{D}$ that is used for training.

For comparison, we also fit a simple 2-dimensional Gaussian model. In this model, each class is assumed to be normally distributed, and a 2-dimensional Gaussian model is fit to each class using maximum-likelihood estimation. Then, we classify points in the corresponding test set by computing the probability that they are generated by the two normal distributions and by picking the class that maximizes this probability. Over the 10 training/test sets, the Gaussian model gives a classifier with mean Precision 47.90%, standard deviation 0.91, mean Recall 87.47%, standard deviation 1.31. This performance is comparable to a particular choice of parameters of SVM to obtain high Recall and low Precision. Computational experiments (not reported in detail) demonstrate that using the Gaussian model leads to weaker bound tightening compared to SVM, with no saving of CPU time on average.

Based on these results, we use an SVM model trained with a polynomial kernel of degree 4, $\lambda = 4$, $\beta = 4$, $\omega = 1$ for the experiments in the remainder of this section; this model has 580 support vectors yielding fast classification. Note that 580 is almost half the size of the training set, suggesting that some overfitting might occur. However, the model shows good performance on examples not included in the training set.

5.3 Testing the probing algorithm

In this section, we discuss the effect of applying Aggressive Probing on a set of difficult MINLPs. In addition to FBBT, we also use Optimality-Based Bound Tightening (OBBT) [12]. This bound-tightening technique maximizes and minimizes the value of each variable over the convexification computed by `Couenne` at the root node, and uses the optimal values as variable bounds. For each test instance, we first apply FBBT and OBBT. Then, for each variable, we apply Aggressive Probing to tighten both the lower and upper bounds, with a time limit of 60 seconds per variable, and 36 hours per instance. The parameter `max_failures` is set to 10. Variables are processed in the order in which they are stored in `Couenne`. Note that `Couenne` uses a standardized representation of the problem where extra variables, called *auxiliary* variables, are typically

added to represent expressions in the original problem formulation [5]. In our experiments, to limit CPU time, OBBT and Aggressive Probing are applied only to original variables; in principle, both can be applied to auxiliary variables without modification.

After Aggressive Probing has been applied to all of the original variables or the global time limit is reached, we record the fraction of tightened variables η, and the average bound reduction ρ, as described in Section 4.1. Then, we apply an additional iteration of FBBT to propagate the new bounds and generate convexification inequalities. This gives a strengthened convexification $\mathcal{C}'$ of $\mathcal{P}$ that is compared to the initial one, $\mathcal{C}$. We record the fraction of variables for which at least one bound could be tightened in $\mathcal{C}'$, as well as the average bound reduction ρ of the tightened variables. Additionally, we compute the percentage of the optimality gap of $\mathcal{C}$ that is closed by $\mathcal{C}'$, i.e., $(z(\mathcal{C}') - z(\mathcal{C}))/(z(\mathcal{P}) - z(\mathcal{C}))$, where $z(\mathcal{C})$ is the optimal objective value of $\mathcal{C}$, and $z(\mathcal{P})$ is the value of the best known solution for the particular instance. The value of $z(\mathcal{P})$ for each instance was obtained from the MINLPLib website.

Results are reported in Table 1. The fraction of tightened variables is relative to the number of original variables for "Probing". For "Probing + FBBT + Conv.", it is relative to the total number of variables, because auxiliary variables can also be tightened after bound propagation through FBBT. The fraction of tightened variables takes into account variables with finite bounds only. Infinite variable bounds are tightened to a finite value only for the three `water` instances, independent of whether SVM is used.

First, we discuss the effect of Aggressive Probing alone. Table 1 shows that the effect of probing is problem-dependent; for example, for `lop97icx`, no variable is tightened by our algorithm, and for `nvs23` and `nvs24`, more than 90% of the variables are tightened. On average, approximately 25% of the original variables are tightened by Aggressive Probing, and after applying FBBT, approximately 30% of the total number of variables (original plus auxiliary) gained tighter bounds. The average bound reduction is close to 50%. The amount of optimality gap closed by adding convexification inequalities after tightening the bound is largely problem dependent as well. The new convexification is much stronger for the `water`, `nvs` and `csched2` instances, but for the remaining instances, the optimality gap is unchanged. This is probably due to the geometry of the initial convexification, for which the LP solution is extremely difficult to cut off without branching, so that no optimality gap is closed by Aggressive Probing. In summary, on all but one test instance, Aggressive Probing is able to provide better variable bounds compared to traditional bound-tightening techniques (FBBT followed by OBBT). This comes at a large computational cost, but may be worth the effort for some difficult instances that cannot be solved otherwise, or when parallel computing resources provide a large amount of CPU power.

Comparing the Aggressive Probing algorithm with and without SVM for failure prediction, we observe on average 30% of computing time saving when using SVM, while the number of tightened variables and average bound

	# s-prob. iter.	
	Success	Failure
Without SVM	1600	18131
With SVM	1634	8998

Table 2. Number of successful and failed s-probing iterations recorded by applying Aggressive Probing on the full test set of Table 1.

tightening is only slightly weaker. CPU time savings are problem dependent: the difference can be huge (`csched2a`, `lop97icx`), or negligible. In only two cases (`nsv24` and `tln6`), using SVM for failure prediction results in an overall longer probing time, but the increase is negligible. Summarizing, using an SVM model to predict likely failures of the Aggressive Probing algorithm leads to CPU time savings that depend on the problem instance at hand and are sometimes very large, sometimes moderate, while variable bounds are tightened by almost the same amount.

Table 2 reports the total number of successful and failed s-probing iterations performed over all test instances. The use of an SVM classifier decreases the number of failed s-probing iterations by a factor two, and increases the percentage of successful s-probing iterations from 8% to 15%. These improvements come at essentially no cost.

5.4 Branch-and-Bound after probing

The main purpose of a bound-tightening technique is to improve the performance of a Branch-and-Bound search. In this section, we report Branch-and-Bound experiments with and without Aggressive Probing on a few selected instances. Table 1 indicates that the probing algorithm proposed in this paper may be effective on the three `water` instances. Therefore, we execute the Branch-and-Bound algorithm of `Couenne` on these instances with a time limit of 24 hours, using the variable bounds obtained after applying FBBT and OBBT at the root node. Then we perform the same experiment using the variable bounds provided by Aggressive Probing with SVM for failure prediction. Results are reported in Table 3, where we include the time spent by probing in the total CPU time.

The `water4` instance is solved with and without Aggressive Probing; Branch-and-Bound without probing is 30% faster, but it explores 20 times as many nodes. Thus, probing is very effective in reducing the size of the enumeration tree. The `waterx` instance remains unsolved after 24 hours. However, employing Aggressive Probing yields a much better lower bound when the time limit is reached (we close an additional 37% of optimality gap). Finally, `waterz` is not solved by Branch-and-Bound unless Aggressive Probing is used. Due to tighter variable bounds, we can solve the instance to optimality in approximately 12 hours, whereas it is unsolved in 24 hours (with 1.2 million active nodes and 23% optimality gap left) if Aggressive Probing is not employed. To the best of our knowledge, an optimality certificate for the solutions to the

	Without Aggr. Probing			With Aggr. Probing + SVM				
	Final			Probing Root		Final		
Instance	Gap %	Nodes	Time	Gap %	Time	Gap %	Nodes	Time
`water4`	100.00	1751046	20252.1	50.47	18013.8	100.00	88902	28977.1
`waterx`	30.31	589477	86415.6	34.83	7874.8	67.11	582046	86393.3
`waterz`	77.78	11088079	86399.6	29.35	18617.8	100.00	1033027	45082.8

Table 3. Results on `water` instances with Branch-and-Bound, with and without Aggressive Probing. We report the percentage of optimality gap closed at the end of the optimization process, the number of nodes, and the total CPU time. When Aggressive Probing is used, we additionally report the optimality gap closed by probing at the root (after recomputing the convexification) and the corresponding CPU time.

`water4` and `waterz` has not been provided previously; the optimal solutions of these instances have objective values 910.8821 and 907.0169, respectively.

6 Conclusions

In this paper, we presented a bound-tightening technique for MINLP that uses truncated Branch-and-Bound searches. Computational tests demonstrate that our Aggressive Probing algorithm is able to tighten the variable bounds on many instances, even after other bound-tightening techniques have been applied. Because Aggressive Probing can easily be carried out in parallel, it is well-suited for leveraging parallel environments to solve difficult MINLPs .

The main drawback of the method is its large computational cost. By using an SVM classifier, we predict success or failure of each iteration of Aggressive Probing, where the prediction is based on features of the algorithm's input. Skipping the probing iterations likely to fail saves on average 30% of the computing time. Currently, the prediction is based on only two easy-to-compute features of the input problem, but we plan to extend the model by taking into account more features, such as the maximum allowed time for the probing iteration, and a measure of its effectiveness from previous iterations.

Computational experiments demonstrate that, on some instances, the overall Branch-and-Bound search is improved using our bound-tightening algorithm. In particular, we are able to solve a difficult MINLPLib instance, `waterz`, for the first time, and we obtain better lower bounds (or a smaller enumeration tree) for other instances.

References

1. Biegler, L., Grossmann, I., Westerberg, A.: Systematic Methods of Chemical Process Design. Prentice Hall, Upper Saddle River (NJ) (1997)
2. Floudas, C.: Global optimization in design and control of chemical process systems. Journal of Process Control **10** (2001) 125–134
3. Tawarmalani, M., Sahinidis, N.: Exact algorithms for global optimization of mixed-integer nonlinear programs. In Pardalos, P., Romeijn, H., eds.: Handbook of Global Optimization. Volume 2. Kluwer Academic Publishers, Dordrecht (2002) 65–86
4. Tawarmalani, M., Sahinidis, N.: Global optimization of mixed integer nonlinear programs: A theoretical and computational study. Mathematical Programming **99** (2004) 563–591
5. Belotti, P., Lee, J., Liberti, L., Margot, F., Wächter, A.: Branching and bounds tightening techniques for non-convex MINLP. Optimization Methods and Software **24**(4-5) (2008) 597–634
6. Savelsbergh, M.W.P.: Preprocessing and probing techniques for mixed integer programming problems. ORSA Journal on Computing **6**(4) (1994) 445–455
7. Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning **20** (1995) 273–297
8. Hutter, F., Hoos, H., Leyton-Brown, K.: Automated configuration of mixed integer programming solvers. In: Proceedings of CPAIOR 2010. Volume 6140 of Lecture Notes in Computer Science. Springer (2010) 186–202
9. Markót, M.C., Schichl, H.: Comparison and automated selection of local optimization solvers for interval global optimization methods. Technical report, Faculty of Mathematics, University of Vienna
10. Sahinidis, N.: Baron: Branch and reduce optimization navigator, user's manual, version 4.0. http://archimedes.scs.uiuc.edu/baron/manuse.pdf (1999)
11. Belotti, P.: Couenne: a user's manual. Technical report, Lehigh University (2009)
12. Shectman, J., Sahinidis, N.: A finite algorithm for global minimization of separable concave programs. Journal of Global Optimization **12** (1998) 1–36
13. Smith, E.: On the Optimal Design of Continuous Processes. PhD thesis, Imperial College of Science, Technology and Medicine, University of London (October 1996)
14. Cristianini, N., Shawe-Taylor, J.: An Introduction to Support Vector Machines and Other Kernel-based Learning Methods. Cambridge University Press, Cambridge, UK (2000)
15. Schölkopf, B., Smola, A.J.: Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond. MIT Press, Cambridge, MA (2002)
16. Achterberg, T., Koch, T., Martin, A.: Branching rules revisited. Operations Research Letters **33**(1) (2005) 42–54
17. Chang, C.C., Lin, C.J.: LIBSVM: a library for support vector machines. (2001) Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.
18. Bussieck, M.R., Drud, A.S., Meeraus, A.: MINLPLib — a collection of test models for Mixed-Integer Nonlinear Programming. INFORMS Journal on Computing **15**(1) (2003)
19. CMU-IBM: Cyber-Infrastructure for MINLP `http://www.minlp.org`.
20. Koggalage, R., Halgamuge, S.: Reducing the number of training samples for fast support vector machine classification. Neural Information Processing - Letters **2**(3) (2004) 57–65