# Shared-Memory Parallel Algorithms for Mixed-Integer Nonlinear Optimization

*A Thesis*
*Submitted in partial fulfillment of*
*the requirements of the degree of*
***Doctor of Philosophy***
*by*

**Prashant Palkar**
(Roll No. 134190002)

Supervisor:
**Prof. Ashutosh Mahajan**


Industrial Engineering and Operations Research
Indian Institute of Technology Bombay
Mumbai 400076 (India)

September 26, 2022

# Acceptance Certificate

**Industrial Engineering and Operations Research**
**Indian Institute of Technology, Bombay**

The thesis entitled "Shared-Memory Parallel Algorithms for Mixed-Integer Nonlinear Optimization" submitted by Prashant Palkar (Roll No. 134190002) may be accepted for being evaluated.

Date: September 26, 2022

Prof. Ashutosh Mahajan

# Approval Sheet

This thesis entitled "Shared-Memory Parallel Algorithms for Mixed-Integer Nonlinear Optimization" by Prashant Palkar is approved for the degree of Doctor of Philosophy.

_____

_____

_____

Examiners

_____

_____

_____

Supervisor (s)

_____

Chairman

Date: _____

Place: _____

# Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I declare that I have properly and accurately acknowledged all sources used in the production of this report. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Prashant Palkar

Date: September 26, 2022 (Roll No. 134190002)

# Abstract

Mixed-integer nonlinear programming problems (MINLPs) are optimization problems characterized by nonlinear functions in constraints and/or objective and integer variables. MINLPs are used to model a wide range of applications in science, engineering, economics, etc. Yet, MINLPs are difficult to solve. We study shared-memory parallel algorithms for MINLPs. These algorithms exploit parallel processing capabilities of modern shared-memory computing architectures.

First, we focus on a specific class of MINLPs called convex MINLPs. We implement shared-memory parallel versions of two well known convex MINLP algorithms: the NLP based branch-and-bound and the LP/NLP based branch-and-bound. These parallel algorithms are implemented within an open-source MINLP toolkit, Minotaur. Our implementations solve different nodes of the branch-and-bound tree concurrently. We analyze the performance of our algorithms using convex instances from the MINLPLib benchmarking library. Our computational results show about 40% improvement in the solution times and an increase in the number of instances solved by using up to 16 cores. These parallelization methods are compared to alternate approaches that exploit parallelism in existing commercial integer linear programming solvers.

Furthermore, we study conditions that may cause a parallel tree-search algorithm to sometimes run slower than its sequential counterpart. Such a phenomenon is called a detrimental anomaly. In the context of convex MINLP algorithms, we present nondetrimental parallel extensions of the NLP based branch-and-bound and the LP/NLP based branch-and-bound algorithms. We recommend settings for our implementations in Minotaur which yield deterministic runs for the proposed parallel algorithms.

We also present a heuristic for nonconvex MINLPs based on a branch-and-bound framework. At each node in the search tree, we solve a continuous nonlinear relaxation multiple times. Since the relaxation we create is in general not convex, this method does not guarantee finding an optimal solution. In order to

obtain good solutions, we solve the relaxation multiple times in parallel starting from different initial points. Our computational experiments show that this approach yields optimal or near-optimal solutions on benchmark MINLP problems, and that the method benefits well from parallelism.

Finally, we study another specific class of discrete optimization problems, called Mixed-Integer Derivative-Free Optimization (MIDFO) problems, in which the mathematical form of the nonlinear functions and their derivatives are not available. We address the problem of minimizing a convex function on a nonempty, finite subset of the integer lattice when the function cannot be evaluated at noninteger points. We propose a new underestimator that does not require access to (sub)gradients of the objective function - such information is unavailable when the objective is a blackbox function. Rather, our underestimator uses secant linear functions that interpolate the objective function at previously evaluated points. These linear mappings are shown to underestimate the objective function in disconnected portions of its domain. Therefore, the union of these 'conditional cuts' provides a nonconvex underestimator of the objective. We propose an algorithm that alternates between updating the underestimator and evaluating the objective function. We prove that the algorithm converges to a global minimum of the objective function on the feasible set. We present two approaches for representing the underestimator and compare their computational effectiveness. In the first approach, we model the underestimator as an MILP that seems difficult for the application we consider. This MILP can be solved in parallel by existing MILP solvers. The second, numerically more robust, approach explicitly maintains lower bounds at integer points and involves a lot of independent computations. Hence, we exploit parallelism within this algorithm too. We also compare implementations of our algorithm with existing MIDFO methods. We discuss the noticeable difficulty of this problem class and provide insights into why a computational proof of optimality is challenging even for moderate problem sizes.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Introduction

Mixed-Integer Nonlinear Optimization problems refer to a class of mathematical optimization problems that have a nonlinear objective function or constraints, along with integer constrained variables. Since an optimization model is often called a 'program', we refer to these models as MINLPs: Mixed-Integer Nonlinear Programs. We use the term MINLP to also refer to Mixed-Integer Nonlinear Programming.

MINLP provides a powerful framework to model and solve a wide range of optimization problems in various applications. However, these problems fall in the category of some of the most difficult optimization problems to solve practically. The reason why MINLPs are so challenging to solve is two-fold. The first reason is the presence of decisions that are discrete in nature, for example, a switch can either be on or off, or the number of vehicles can only be an integer value like 5 or 6, but not 5.5. For most practical cases, this discrete nature of multiple decisions eventually results in a prohibitively large number of combinations (of decisions) to be either evaluated, or ruled out in lieu of an available good solution. Modelling such restrictions rules out many available mathematical frameworks like Linear Programming that boast of practically efficient solution methods. The other reason why MINLPs are difficult to solve is the presence of nonlinearity in the constraints or the objective function. Only a few special cases of a system of nonlinear inequalities are known to be solvable efficiently.

Even with the latest tools and techniques, finding even one solution to a given numerical instance of a MINLP may require hours or days on the most sophisticated computer systems. Therefore, these problems have been the centre of theoretical and computational research. On the computational front, recent advances in computing systems, especially the parallel computing architectures,

enable a user to execute multiple tasks simultaneously. Personal computers and workstations that are commonly used today are equipped with technologies that enable parallel processing. Most of them have multiple processors that share a common memory space for performing their tasks; such systems are referred to as shared-memory parallel systems. This thesis focusses on fast practical methods for solving MINLPs using multiple processors available on shared-memory parallel computing environments.

## 1.1 MINLP and Related Problems

A MINLP can be mathematically expressed as:

$$\begin{aligned} &\underset{x}{\text{minimize}}\ f(x) \\ &\text{subject to } g(x) \leq b, \\ &\qquad\qquad x \in \mathcal{X}, \\ &\qquad\qquad x_j \in \mathbb{Z}, \quad \forall j \in \mathcal{I}. \end{aligned} \tag{P}$$

Here, $x = (x_1, \ldots, x_n)^\top$ represents the vector of decision variables, some of which are restricted to take only integer values. The set of indices of integer-constrained variables is denoted by $\mathcal{I}$. The constraints on the decision variables are expressed using inequalities, with a vector of real values $b = (b_1, b_2, \ldots, b_m)^\top$ on the right hand side and the functions $g : \mathbb{R}^n \to \mathbb{R}^m$ on the left hand side. We initially assume that the functions $g_1, g_2, \ldots, g_m$ are nonlinear and twice continuously differentiable, and later consider problems where derivatives are not available. The relation $g(x) \leq b$ represents a set of $m$ nonlinear inequality constraints. An equality constraint of the form $\bar{g}(x) = \bar{b}$ can be rewritten as a pair of inequalities, $\bar{g}(x) \leq \bar{b}$ and $-\bar{g}(x) \leq -\bar{b}$. The set $\mathcal{X}$ represents the collection of linear constraints in the problem. Precisely, $\mathcal{X} := \{x : Cx \leq c,\ Dx = d\}$ where $C$, $D$ are matrices and $c, d$ are vectors of appropriate dimensions. We assume that the set $\mathcal{X}$ is bounded. $f : \mathbb{R}^n \to \mathbb{R}$ represents the objective function of the optimization problem (P) that one wants to minimize while satisfying all the constraints on the decision variables $x$.

The MINLP framework is used in modeling optimization problems arising in various scientific, engineering, economic and managerial applications. Applications include portfolio optimization, facility location problems, process design, unit commitment, water and gas networks design, cutting stock problem, protein folding, etc. Other more recent ones include brachytherapy, cyber security,

energy management, cloud computing, supercomputing, environment, weapons target assignment, etc. The details on these applications and the references involving the respective MINLP formulations can be found in the surveys by **?**, **?**, **?**, **?**, **?**, etc. MINLPs have also been used as subproblems within more general optimization frameworks like in derivative-free optimization (**??**), partial differential equation constrained optimization (**?**), bilevel optimization (**?**), etc. A nice review of such domains and their intersection with MINLP can be found in the surveys by **?** and **?**.

An important classification of MINLP is based on the 'convexity' of the nonlinear functions $g$ and $f$ in (P). A convex function is formally defined as follows.

**Definition 1.1.1** (Convex function)**.** *(**?**) Let $C \subseteq \mathbb{R}^n$ be a nonempty convex set. A function $f : C \to \mathbb{R}$ is called a convex function if and only if $f\left(\lambda x^1 + (1-\lambda)x^2\right) \leq \lambda f(x^1) + (1-\lambda) f(x^2)$ for all $x^1, x^2 \in C$ and some scalar $\lambda \in [0,1]$.*

Another definition of convex functions when they are differentiable is as follows.

**Definition 1.1.2** (Convex function)**.** *(**?**) Let $C \subseteq \mathbb{R}^n$ be a nonempty open convex set. Suppose that a function $f : C \to \mathbb{R}$ is differentiable, that is, its gradient $\nabla f$ exists at all points in $C$. Then, $f$ is convex if and only if,*

$$f(x^2) \geq f(x^1) + \nabla f(x^1)^\top (x^2 - x^1), \tag{1.1}$$

*for any $x^1, x^2 \in C$.*

The affine function on the right hand side in (1.1) is the first order Taylor's approximation of $f$ near the point $x^1$. This property of convex functions enables one to use local information about a convex function (its value and derivative at a given point, $x^1$) to derive a global underestimator, a linear function in this case (over the entire domain, $C$).

In the context of MINLP, if the functions $g_1, g_2, \ldots, g_m$ and $f$ are convex, then we call (P) a convex MINLP. MINLPs that are convex are less difficult than the ones those are not convex (also referred to as nonconvex MINLPs). The set of feasible points of a convex MINLP is shown in Figure 1.1. Property 1.1 is widely exploited in algorithms for convex MINLP.

There are some well studied classes of optimization problems that can be considered as special cases of MINLPs. We describe three of them: Nonlinear Programs (NLPs), Mixed-Integer Linear Programs (MILPs) and Linear Programs (LPs). Algorithms for MINLP solve a sequence of related NLPs, MILPs and LPs.

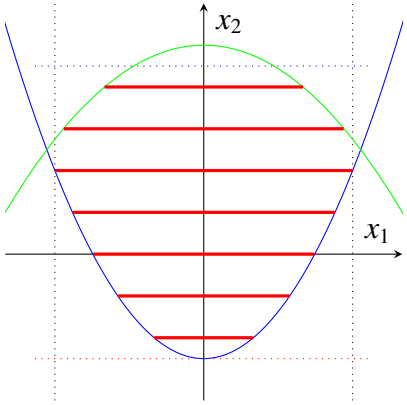



**Figure 1.1:** A convex mixed-integer nonlinear programming problem in two dimensions. The red lines indicate the points that satisfy all the constraints of (P), called the feasible region of this MINLP, and the (convex) nonlinear constraints are shown using the curved lines. One can note that the variable $x_2$ is integer constrained.

### 1.1.1 Nonlinear Programs

If the integer restrictions on decision variables $x$ are relaxed in (P), then we obtain an NLP of the form:

$$\begin{aligned} &\underset{x}{\text{minimize}}\ f(x) \\ &\text{subject to } g(x) \leq\ b, \\ &\qquad x \in \mathcal{X}. \end{aligned} \tag{R}$$

In the absence of constraints $g$ and the case when $\mathcal{X} = \mathbb{R}^n$, the problem would be called an 'unconstrained' minimization problem (that is, there are no constraints on the decision variables $x$). Otherwise, we call the problem (R) a 'constrained' NLP. If $g$ and $f$ are convex differentiable functions, then we call (R) a convex NLP. Convex NLPs are considered 'tractable', which means that there are methods that can solve such problems in a reasonable amount of time. Formally, most of these problems fall under the so called complexity class $\mathcal{P}$ (**?**) of problems solvable in polynomial time. Informally, this means that there exists a method that can solve all numerical instances of this problem in a reasonable amount of time. In the absence of nice properties like convexity on $f$ and $g$, (R) is also difficult to solve.

Next, we differentiate a 'local' and a 'global' optimal solution of (R). A feasible solution, say $\bar{x}$, of an optimization problem is a point that satisfies all the constraints in the problem. Let us denote the set of feasible solutions of (R) by $\mathcal{F}_{(R)}$. An optimal solution, say $x^*$, is a feasible solution that yields the best possible value of the objective function $f$. Formally,

**Definition 1.1.3** (Global minimizer). *A point $x^* \in \mathcal{F}_{(R)}$ is called a global minimizer of (R) if $f(x^*) \leq f(\bar{x})$ for all $\bar{x} \in \mathcal{F}_{(R)}$.*

For a general NLP of the form (R), it is difficult to find or verify a global minimum. Hence, the notions of 'neighbourhood' and local minimum are used. A neighbourhood of a point $\bar{x}$ is the set of the points in a radius $\epsilon$ of $\bar{x}$, defined as, $\mathcal{N}(\bar{x}, \epsilon) := \{x : \|x - \bar{x}\| < \epsilon\}$ for a given small scalar $\epsilon$.

**Definition 1.1.4** (Local minimizer). *A point $x^*$ is called a local minimizer of (R) if there exists a scalar $\epsilon > 0$ such that $f(x^*) \leq f(\bar{x})$ for all $\bar{x} \in \mathcal{F}_{(R)} \cap \mathcal{N}(\bar{x}, \epsilon)$.*

A local minimizer is a global minimizer for convex problems.

**Theorem 1.1.5.** *(?, Section 4.2.2) Let (R) be a convex optimization problem. If $x^*$ is a local minimum of (R), then it is also a global minimum of (R).*

### 1.1.2 Mixed-Integer Linear Programs

Another important subclass of MINLPs is the Mixed-Integer Linear Programs (MILPs), mathematically represented as

$$\begin{aligned} \underset{x}{\text{minimize}}\ & f(x) \\ \text{subject to}\ & Cx \leq c, \\ & Dx = d, \\ & x_j \in \mathbb{Z},\ \forall j \in \mathcal{I}, \end{aligned} \tag{M}$$

where $f$ is a linear function. It may be noted that due to the presence of integer constrained decision variables, the feasible region of (M) is not convex. MILPs are known to be generally difficult to solve and in fact belong to the class of $\mathcal{NP}$-hard problems (?). It means that the computational effort to solve these problems increases exponentially in the size of the input (of problem instance).

### 1.1.3 Linear Programs

A closely related class of problems to MILPs is the Linear Programs (LPs), mathematically represented as

$$\begin{aligned} \underset{x}{\text{minimize}}\ & f(x) \\ \text{subject to}\ & Cx \leq c, \\ & Dx = d, \end{aligned} \tag{LP}$$

where $f$ is again a linear function. Amongst the problems described so far, LPs are the easiest to solve. Formally, they belong to the complexity class $\mathcal{P}$, the class of problems for which efficient algorithms that run in time polynomial in the size of the input, are known (**?**).

We next explain the importance of the above three classes of problems in the context of convex MINLP.

## 1.2 Algorithms for Convex MINLP

MINLP is $\mathcal{NP}-$hard, because MILP, which is a special case of MINLP is $\mathcal{NP}-$hard. In this section, we describe deterministic algorithms for solving convex MINLPs, all of which are based on a 'branch-and-bound' framework. First, we present an important notion of a relaxation that is used in these algorithms.

**Definition 1.2.1** (Relaxation)**.** *(**?**) A relaxation of the problem (P) is another problem that either has a (bigger) feasible region enclosing the feasible region of (P), or has an objective function that underestimates the objective function $f$ of (P) at all points in the feasible region of (P), or both.*

It is easy to see that the optimal objective function value of a relaxation would always be less than or equal to that of the original problem, say $z^*$. Thus, the optimal value of a relaxation provides a 'lower bound' on $z^*$. A relaxation is considered 'tight' when its feasible region closely approximates that of the original problem or when its objective function value is not too far from the original objective function at (points near) an optimal solution. Methods for solving convex MINLPs primarily differ in the way they create a relaxation of the MINLP. For practical purposes, generally one wants to construct tight relaxations that are also easier to solve compared to the original problem.

### 1.2.1 Nonlinear Branch-and-Bound (NLP-BB)

The nonlinear branch-and-bound method (**?**) for convex MINLP is based on the branch-and-bound framework, just like the one for MILP. A branch-and-bound method starts by solving a relaxation of (P), that has a larger feasible region enclosing (P), but is easier to solve to global optimality. A solution of this relaxation provides a valid lower bound on the optimal value $z^*$ of (P). Then one divides the search-space by 'branching' to create smaller subproblems. A relaxation of each subproblem is then solved. Each subproblem, a smaller relaxation

than its parent, has a lower bound no less than its parent. If a solution to any of the subproblems is feasible for (P), its objective value provides an upper bound on $z^*$. The algorithm stops when the lower and the upper bounds on $z^*$ converge. This setup is easily viewed and analyzed as a tree-search where the nodes denote the subproblems and the edges denote the branches that divide a subproblem.

The NLP-BB algorithm for convex MINLP creates an initial relaxation of (P) obtained by simply removing the integrality constraints on decision variables. This relaxation is a convex NLP of the form (R). A pictorial depiction of (R) is shown in Figure 1.2. Also, an illustration of the tree representation of the branch-

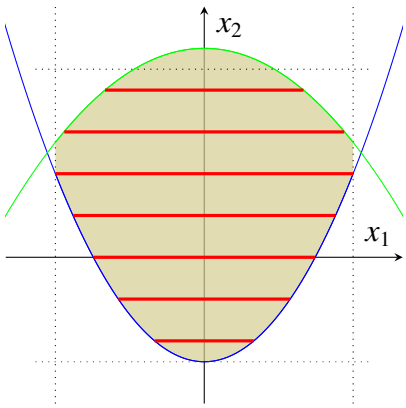



**Figure 1.2:** A nonlinear programming relaxation (the shaded region) of the convex MINLP shown in Figure 1.1, obtained by relaxing the integer restrictions on decision variables.

and-bound is shown in Figure 1.3. We elaborate more on two critical concepts,

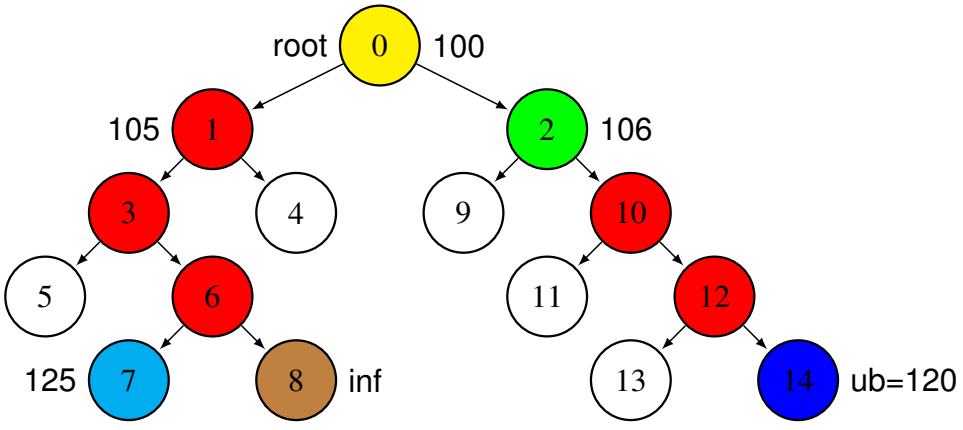



**Figure 1.3:** A tree-based depiction of nonlinear programming relaxations based branch-and-bound (NLP-BB) algorithm for convex MINLP.

'branching' and 'pruning', in the branch-and-bound algorithm. Branching refers to partitioning the search space (or a given part of the feasible region) in such

a way that no feasible solution in that region is excluded. In problems with integer constrained variables, this can simply be done by branching on an integer variable $x_j \in \mathcal{I}$, one that has assumed a nonintegral value, say $x_j^* \notin \mathbb{Z}$ in the solution of a relaxation, to create two subproblems (children nodes in the search tree) using the two inequalities, $x_j \leq \lfloor x_j^* \rfloor$ and $x_j \geq \lceil x_j^* + 1 \rceil$. This process is depicted geometrically in Figure 1.4.

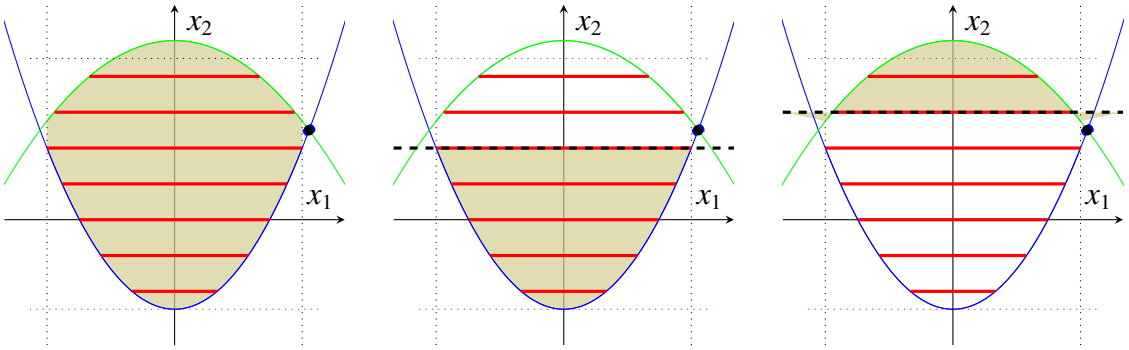



**Figure 1.4:** Geometrical illustration of variable-based branching of a region in NLP-BB algorithm. Here, $x_2$ is the integer constrained variable and is used to divide the search space, shown on the left, into two parts, shown in the centre and on the right.

Pruning makes the branch-and-bound algorithm practically faster than a complete enumeration (of all possible combinations of integer constrained variables) to find an optimal solution. Implicitly, NLP-BB (like other convex MINLP algorithms) tries to avoid parts of feasible region of the relaxation that have no solution, or no better solution than the one already at hand. The former is referred to as pruning by infeasibility, and the latter is termed as pruning by bound. As shown in Figure 1.3, if the relaxation at a tree-node becomes infeasible (node 8), this region can be pruned by infeasibility and need not be explored further. Similarly, if the optimal solution at a relaxation turns out integer feasible (node 14), then this solution is a feasible solution to (P) and its objective value is an upper bound on $z^*$ (here, we have $z^* \leq 120$). This node also can be pruned because we already have the best possible solution of this part of the feasible region. Another possibility is that the lower bound at a node becomes equal to or exceeds the best known upper bound (node 7), in which case, we need not explore this part of the feasible region because it does not contain any solution better than the one at hand, hence, can be pruned by bound.

The NLP-BB algorithm starts working with the NLP relaxation. If relaxation (R) of (P) is infeasible, then so is (P). If the solution, say $x^0$, of the NLP relaxation satisfies integer restrictions of (P), then it is an optimal solution to (P)

as well. If, on the other hand, $x^0$ does not satisfy the integrality restrictions, we get a lower bound on the optimal value of (P). The traditional nonlinear branch-and-bound **?** method proceeds by dividing the search-space into two or more subproblems in a way that every solution of (P) lies in at least one of the subproblems while $x^0$ does not lie in any of them. Each subproblem thus created is a smaller MINLP, and this process is continued recursively.

### 1.2.2 Outer Approximation (OA)

Creating good relaxations that provide a lower bound closer to $z^*$ in a reasonable amount of time is important for fast convergence of the branch-and-bound based algorithms. Rather than starting with a tight relaxation, which may be difficult to obtain or solve, one can first solve a weaker relaxation and then tighten it iteratively by adding 'valid inequalities'. A valid inequality is an inequality constraint that is satisfied by all the feasible points of (P). Combining this scheme with branch-and-bound leads to what is called a branch-and-cut method which most solvers deploy for solving MILPs and MINLPs. A commonly used technique for creating linear relaxations of convex nonlinear constraints is through a gradient based linearization. As mentioned in Definition 1.1.2, given a convex differentiable nonlinear function $\hat{g} : \mathbb{R}^n \to \mathbb{R}$ and a point $x' \in \mathbb{R}^n$, the following gradient inequality (**?**)

$$\nabla \hat{g}(x')^\top (x - x') + \hat{g}(x') \leq \hat{g}(x)$$

holds for all $x \in \mathbb{R}^n$. One can thus create a relaxation of (P) by replacing all of its (convex) nonlinear constraints by

$$\nabla g(x')^\top (x - x') + g(x') \leq b. \tag{Grad-I}$$

Such a relaxation can then be tightened by adding linearization inequalities obtained using multiple points. Two algorithms are based on such linearizations-based relaxations: outer-approximation and the LP/NLP based branch-and-bound algorithm (discussed in Subsection 1.2.3).

The outer-approximation (OA) algorithm proposed by **?** solves an alternating sequence of MILPs and NLPs. It is initialized by solving (R). If a solution $x^0$ of (R) is not integer feasible, the nonlinear functions are replaced by linearization inequalities (Grad-I) obtained at $x^0$, and the integer restrictions are re-introduced to obtain the following MILP relaxation. If the objective function is also nonlinear, the problem is reformulated by replacing the objective with an auxiliary variable,

$\eta$, and adding the constraint $f(x) \leq \eta$. This new constraint is also replaced by its linearization inequality at $x^0$ in the MILP relaxation:

$$\begin{aligned} \underset{x,\eta}{\text{minimize}}\ & \eta \\ \text{subject to}\ & \nabla f(x^0)^\top (x - x^0) + f(x^0) \leq \eta, \\ & \nabla g(x^0)^\top (x - x^0) + g(x^0) \leq b, \\ & x \in \mathcal{X}, \\ & x_j \in \mathbb{Z},\ \forall j \in \mathcal{I}. \end{aligned} \tag{RM}$$

A pictorial illustration of (RM) is shown in Figure 1.5. The MILP relaxation (RM)

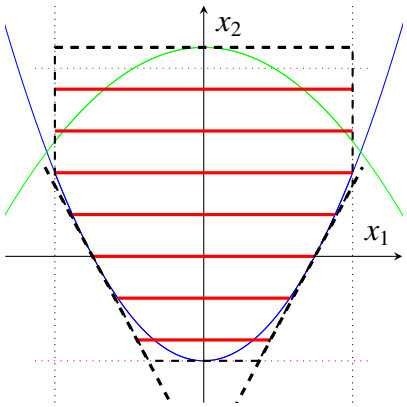



**Figure 1.5:** A mixed-integer linear programming relaxation (the horizontal lines) of the convex MINLP shown in Figure 1.1, obtained by linearizing the nonlinear constraints in (P).

is solved using an MILP solver. If the MILP relaxation is infeasible, then so is (P). If the MILP solution (say, $\hat{x}$) satisfies all nonlinear constraints, then it is optimal to (P). Otherwise, the MILP optimal value (say, $\hat{z}$) provides a lower bound on $z^*$. Next, a 'fixed' NLP of the following form is solved.

$$\begin{aligned} \underset{x}{\text{minimize}}\ & f(x) \\ \text{subject to}\ & g(x) \leq\ b, \\ & x \in \mathcal{X}, \\ & x_j = \hat{x}_j,\ \forall j \in \mathcal{I}. \end{aligned} \tag{F-NLP}$$

We denote this NLP as F-NLP($\hat{x}$) to indicate that the integer variables are fixed to the values in $\hat{x}$. An optimal solution to F-NLP($\hat{x}$) provides an upper bound on $z^*$. The optimal solution is then used to generate more linearization constraints (Grad-I) that are added to the MILP relaxation. The updated MILP relaxation is solved again and the process is repeated. The new inequalities ensure

that all solutions of MILP with $x_j = \hat{x}_j$, $j \in \mathcal{I}$ have objective value no less than $\hat{z}$. If the 'fixed' NLP is infeasible, the point returned by the NLP solvers can still be used to generate valid underestimators and linear constraints (**?**). These linearization inequalities forbid the integer combination $\hat{x}_j$, $j \in \mathcal{I}$ in the future MILP solutions. Another related algorithm, the Generalized Benders Decomposition (GBD) algorithm given by **?**, generates a single inequality at the NLP solution which is then added to the MILP. Both OA and GBD do not require any implementation of tree-search unlike the NLP based branch-and-bound. They naturally exploit the advances that have been made in the MILP technology over the decades, including presolving (**??**), cutting planes (**??**), heuristic search (**??**), conflict analysis (**??**) and parallel search (**???**) etc.

### 1.2.3 LP/NLP Based Branch-and-Bound (QG)

The LP/NLP based branch-and-cut algorithm of **?**, which is also referred to as the QG algorithm, tries to overcome the difficulty of solving similar MILPs repeatedly as in OA method. It creates and maintains a single branch-and-cut tree. Like OA, it starts by solving the NLP relaxation (R), and creates a linear relaxation of (P) by relaxing integrality from (RM), expressed as follows.

$$\begin{aligned} &\underset{x,\eta}{\text{minimize}}\ \eta \\ &\text{subject to } \nabla f(x^0)^\top (x - x^0) + f(x^0) \le \eta, \\ &\qquad\qquad \nabla g(x^0)^\top (x - x^0) + g(x^0) \le b, \\ &\qquad\qquad x \in \mathcal{X}. \end{aligned} \tag{RL}$$

A graphical illustration of the feasible region of (RL) is shown in Figure 1.6.

QG method then initiates the single-tree by solving this root LP relaxation of (P), and proceeds like the LP based branch-and-cut method. When a node in the search-tree yields an integer optimal solution ($\hat{x}$), F-NLP($\hat{x}$) is solved. If the NLP is feasible, its optimal solution provides an upper bound on $z^*$. Linearization inequalities obtained at the point returned by solving F-NLP($\hat{x}$), say $\check{x}$, are added to all the open nodes of the tree to tighten the relaxations, and branch-and-cut is resumed.

## 1.3 Methods for Nonconvex MINLP

Solving nonconvex MINLPs is sometimes referred to as 'global optimization'. Nonconvex MINLP is theoretically 'undecidable' (**?**) but one can find solu-

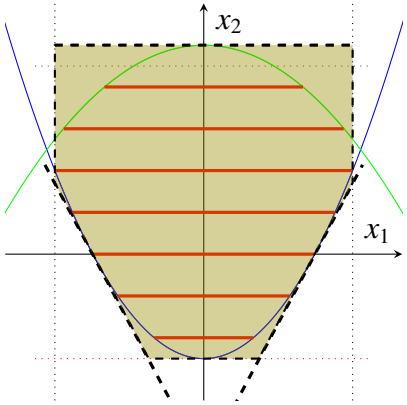



**Figure 1.6:** A linear programming relaxation (the shaded region) of the convex MINLP shown in Figure 1.1, obtained by linearizing the nonlinear constraints and relaxing the integer restrictions on decision variables in (P).

tions when the set $\mathcal{X}$ is bounded. The major difficulty in addressing nonconvex MINLPs is that their continuous relaxations obtained by relaxing integrality constraints, are not tractable (unlike the case of convex MINLPs). Relaxations of nonconvex constraints can be obtained by exploiting special structures in the corresponding nonconvex functions. One way to achieve this is by forming piecewise-linear underestimators of all the nonlinear constraints over the feasible region to obtain an MILP or an LP relaxation. Piecewise linear approximations (**?**) can capture nonconvex functions using a set of breakpoints but practically, this approach is limited to functions with few arguments otherwise the number of pieces needed to get an acceptable approximation can be large.

### 1.3.1 Relaxations Using Factorable Functions

Another way of obtaining tractable relaxations to nonconvex MINLPs is 'convexification' (**??**), that is, to form convex underestimator functions, say $g_i^{cnvx}(x)$, such that $g_i^{cnvx}(x) \leq g_i(x), \forall x \in \mathcal{X})$ , for each nonconvex function $g_i(x), i = 1, \ldots, m$. Replacing each nonconvex constraint then with its convex underestimator yields a convex relaxation. To tighten the relaxation, one can also add concave overestimators of nonconvex constraints, say $g_i^{cncv}(x)$, such that $-g_i^{cncv}(x) \leq -g_i(x), \ \forall x \in \mathcal{X}$.

Convex underestimators of multivariate nonconvex functions are not easy to obtain though. One way is to first reformulate the original nonconvex constraints using simple univariate functions that are simpler to analyze and relax. Using the structure and form of the involved functions in such cases is important. One such reformulation is based on the 'separability' of the functions. A function $\bar{g}(x)$

is termed separable if there exist univariate functions $h_j(x_j)$ such that

$$\bar{g}(x) = \sum_{j=1}^{n} h_j(x_j).$$

Separable functions are quite effective when convex underestimators for the functions $h(x_j)$ are known, because an overall underestimator of $\bar{g}(x)$ is easy to obtain. This is because the sum of underestimators itself is an underestimator of $\bar{g}(x)$. In the absence of properties like separability, one attempts to dissect the nonlinear functions as the sums and products of finitely many univariate functions. Such functions are called 'factorable' functions. These functions can be expressed using operators $+, -, *, /, \sin, \cos, \log, \exp$, etc., with variables or constants as their arguments. One reformulation of such problems can be done by introducing 'auxilliary' variables and related sets of constraints that involve only univariate functions. As an example, consider the constraint

$$x_1 x_3 + x_2^2 + \sin(x_3) \leq 5,$$

where $1 \leq x_1 \leq 3, -2 \leq x_2 \leq 4$ and $0 \leq x_3 \leq 5$. A reformulation of this constraint using new auxilliary variables $y_1, y_2$ and $y_3$ would be the set of constraints:

$$\begin{aligned}
&y_1 + y_1 + y_3 \leq 5,\\
&y_1 = x_1 x_3,\\
&y_2 = x_2^2,\\
&y_3 = \sin(x_3),\\
&0 \leq y_1 \leq 15, 0 \leq y_2 \leq 16, 0 \leq y_3 \leq 1.
\end{aligned}$$

The bounds on the auxilliary variables have been derived based on the related function and the bounds of original variables. After such a reformulated problem is obtained, the sets representing the nonconvex equality constraints, each involving a simple univariate or bivariate function, are replaced by their corresponding convex or polyhedral 'envelops'. Finding tight envelops even for simple sets is a difficult task, although a few sets that appear commonly in many formulations have been studied. For example, a polyhedral relaxation of bilinear sets (like those of the form $y_1 = x_1 x_3$) is given by **?**. A polyhedral LP relaxation or a convex NLP relaxation is then finally obtained when the integer restrictions are also relaxed.

One can also use other techniques to obtain tractable relaxations like $\alpha$-convexification (used in $\alpha$-BB algorithm by **?**) or semidefinite relaxations (**?**).

### 1.3.2 Spatial Branch-and-Bound (SBB)

Once a tractable relaxation is obtained for a nonconvex MINLP, one can get a valid lower bound on its optimal objective function value. It can then be used in a branch-and-bound framework along with a mechanism for branching, in a way similar to the NLP-BB algorithm explained in Section 1.2. Standard branching (on integer variables) can be done when a subproblem node does not yield an integer feasible solution. However, branching on continuous variables (called 'spatial branching') may be necessary if a nonlinear constraint is violated by the optimal solution (say $\hat{x}$) of the relaxation. Spatial branch-and-bound is a widely used algorithm for solving nonconvex MINLPs: a branch-and-bound algorithm that uses spatial branching. As in standard branching, choice of the branching variable is important in spatial branching too in the sense that the subproblems generated must be as tight as possible and must exclude the solution of the previous relaxation solved. Also, unlike convex MINLPs, the convergence of SBB does not follow from the finite number of the integer feasible points. For SBB to converge, finiteness of the bounding operation is required.

## 1.4 Heuristic Approaches

Finding an optimal solution to a MINLP may often be difficult and time consuming. Exact approaches may sometimes fail to find even one feasible solution in a reasonable amount of time. One can use specialized algorithms called heuristics that are designed to provide a feasible solution quickly. Finding a good solution early in a branch-and-bound framework can prove beneficial in accelerating the convergence, especially by pruning inferior nodes in the search tree.

Many heuristics from MILPs have been extended to MINLPs. Some notable ones are Feasibility Pump (**??**), Diving heuristics (**?**), Undercover (**?**), etc. For an extensive survey of MINLP heuristics, we refer the interested readers to **?** and **?**. Exact algorithms for convex MINLPs can also be used as heuristics for nonconvex MINLP as algorithms for convex MINLP usually terminate faster than the exact algorithms for nonconvex MINLPs.

## 1.5 Mixed-Integer Derivative-Free Optimization

If the functions $f$ and $g$ in (P) are not available in a 'compact' mathematical form, and their derivatives are also not available, we refer to such a problem as a Mixed-

Integer Derivative-free Optimization (MIDFO) problem. Many real-world problems from science, engineering and economics applications result in optimization problems that involve the so-called 'black-box' functions. These functions are characterized by the fact that a single function evaluation is often computationally expensive, for example, requiring to run a complex simulation. Also, derivatives of these functions can not be computed or estimated using finite differences, etc., efficiently. Such problems are called Derivative-free Optimization (DFO) problems. When such problems also involve integer constrained variables, then they are referred to as MIDFO problems.

Exact methods like those for MINLPs can rarely be applied to MIDFO problems, due to the absence of derivative information. Also, simple explorative methods such as local search (**?**), variable neighbourhood search (**?**), etc., that look for better points in a local neighbourhood of a given point, or (stochastic) metaheuristics like genetic algorithms, particle swarm algorithms or simulated annealing, etc. (**?**), that generate an improved set of iterates from the previous ones, are not suitable for MIDFO problems, because they require a lot of function evaluations. Therefore, algorithms for MIDFO are designed to minimize the objective function $f$ by exploring the integer feasible points effectively, while limiting the number of function evaluations within a given budget.

A detailed description of the concepts of DFO can be found in the textbooks by **?** and **?**. A review of applications of MIDFO problems, algorithms and software can be found in the surveys by **?**, **?** and **?**.

In a derivative-free setting with discrete sets, it is difficult to certify optimality of a given integer point (as complete enumeration is prohibitive). Therefore, algorithms for MIDFO are designed to terminate at some 'stationary' points, those that satisfy some notion of local optimality in some well defined discrete neighbourhood on the integer lattice. For example, a 'mesh-isolated minimizer' is a point that yields the lowest function value among its neighbouring integer points. A description of different such local optima is given by **?**. Most algorithms for MIDFO are adaptations of (continuous) DFO algorithms to cater to integer constrained variables, for example, using rounding, sets of discrete or integer search directions, branch-and-estimate, etc. Deterministic algorithms for DFO can be classified into two main categories: direct search methods and model-based methods. These methods are briefly described next. Methods for MIDFO are described in Chapter 5.

### 1.5.1 Direct Search Methods

Direct search methods are simple sampling methods that decide upon the next trial iterate or a set of iterates based on a predefined strategy. The classical algorithm by **?** samples a set of points that form a 'simplex' in each iteration. For example, in $\mathbb{R}^2$, a set of three affinely independent points form a triangular simplex. The algorithm starts by sampling such a set. It then proceeds by attempting to replace the point with the worst function value by another point (chosen using some geometric operations like reflection, expansion or contraction, etc.) that yields a new simplex. Another direct search method is the generalized pattern search (**?**) that works using a pattern of points and constitutes mainly two steps: the 'search step' and the 'poll step'. The search step searches among a finite set of directions to find an improved iterate. If the search fails, a 'poll step' is executed that generates a set of 'positive spanning' set of directions, that is, a set of directions that is assumed to contain at least one descent direction for the objective function under certain assumptions. Another popular direct search algorithm is the mesh adaptive direct search by (**?**) which basically is similar to pattern search, but uses a poll size parameter and a mesh size parameter, to carefully restrict the region from where the poll points are selected. These algorithms eventually converge when a termination criterion is met, for example, the line search parameter like the mesh size, step size or the simplex diameter is below a certain threshold parameter.

### 1.5.2 Model Based Methods

These methods exploit the available sampling information to initially fit and then improve a surrogate model that approximates the behaviour and properties of the function, and then optimize it to obtain iterates in an intelligent way. The surrogate approximation model is optimized using traditional derivative based methods depending upon the choice of the model. The two key decisions in model based methods are the selection of the appropriate surrogate model and choice of the next sample point or set of points. Some popular models are response surface metamodels that include interpolation methods like krigging (**?**) and radial basis functions (**??**). The models used in these algorithms usually get better with more function evaluations. Model based methods terminate when some convergence criteria like limited improvement in consecutive iterations of the algorithm, etc. are satisfied.

## 1.6 Shared-Memory Parallelism

Since many discrete optimization algorithms like branch-and-bound, particularly for MILPs and MINLPs, break down the overall problem into several independent small subproblems, such algorithms are attractive for parallel computing. Parallel computing architectures comprise multiple processors or CPUs and can be broadly classified into 'shared-memory' or 'distributed-memory' systems. As the name suggests, each processor (or node) in a distributed-memory system has its private memory and typically requires dedicated hardware and networking interfaces to exchange data. Typical distributed memory systems are built to address large-scale problems and have thousands of processors (or more). Supercomputers are a specific example of distributed memory systems. For instance, the fastest supercomputer in India, **?**, has about 84000 GB of memory distributed over 41664 processors.

On the other hand, in a shared-memory architecture, a standard block of memory is available for multiple processing units (also called cores), typically placed on a single integrated circuit. Desktop computers, laptops, mobile phones, etc., very commonly deploy multiprocessing CPUs nowadays. While the Moore's Law and the Dennard scaling law (**?**) predicted a continuing increase in clock speeds of CPUs, the breakdown of the latter in the late 2000s gave impetus to the manufacturing of multicore processors. Many hybrid systems have come up recently, but typical shared-memory systems have a limited number of processing cores per CPU, say 96, for example. However, each processor has access to the memory. Hence the data access times are faster compared to distributed-memory systems.

From the viewpoint of algorithm design, both architectures require a programmer to identify tasks (data) that can be executed (stored) independently, along with those that are inherently sequential and minimize duplicate or unnecessary computing. Shared-memory systems are often easier to program as compared to distributed-memory systems because in distributed-memory systems, one needs to explicitly specify the type and schedule of data exchange between processors (using message passing), whereas in shared-memory systems, all memory is accessible either uniformly or non-uniformly to all the processors. In shared-memory systems, two or more processors may attempt to read/write a shared memory location simultaneously (called memory contention) which may

cause a program to crash or yield faulty performance. Thus, memory-access has to be dealt with carefully.

In this thesis, we will focus on designing algorithms for MINLPs on shared-memory computers. Since shared-memory systems are almost a norm now, parallel algorithms that efficiently exploit such systems would be convenient. Also, with increasing usage of hybrid architectures that combine distributed-memory and shared-memory systems, multi-level parallelization models appear promising (see **?**), where some lower level parallelism would be on shared-memory systems/nodes. In the context of MINLP, branch-and-bound based methods may seem easy to parallelize, as the dynamic generation of subproblems and their processing can be done independently.

## 1.7 Solvers for MINLP and Minotaur

Most solvers can read mathematical models through various file formats produced by popular mathematical modeling software like AMPL (**?**), GAMS (**?**), OPL (**?**), AIMMS (**?**), Pyomo (**?**), Optimization Toolbox in **?**, etc. Optimization solvers cater to the difficult task of deploying the most appropriate data structures, classes, and the most efficient logic that results in robust and fast optimization algorithms. Typically, both commercial and open-source packages provide the users either with the flexibility to customize and experiment with various algorithmic components within existing algorithms or provide frameworks that can be used to implement novel algorithms by plugging in (or out) key algorithmic features. Some of the key algorithmic components in MINLP solvers are presolving, heuristics, branching, cutting planes and node-processing subsolvers (subroutines for solving the subproblems: LPs, NLPs and MILPs).

Since most state-of-the-art algorithms for both MILPs and MINLPs are based on branch-and-bound, many software traditionally developed for MILPs are capable of solving MINLPs like SCIP (**?**), AIMMS (**?**), **?**, etc. Solvers designed basically for MINLPs are categorized into either convex MINLP solvers, for example, BONMIN (**?**), **?**, Minotaur (**?**), or nonconvex MINLP solvers like BARON (**?**), Couenne (**?**), ANTIGONE (**?**), etc. SCIP can solve nonconvex MINLPs, while AIMMS and Gurobi are largely convex MINLP solvers.

Most of the algorithms we present in this thesis in the subsequent chapters have been implemented within the open-source solver Minotaur. We briefly describe the salient features of Minotaur later in Chapter 2.

## 1.8 Motivation for the Thesis and Outline

Parallel approaches have been used mostly to solve continuous problems or MILPs (**??????**).

A few parallel approaches for MINLP have been proposed earlier. **?** proposed running QG and the 'Tabu Search' metaheuristic concurrently using two threads. **?** present NLP-BB on a 'computational grid' (a cluster of geographically distributed computing resources with heterogeneous capabilities) to solve nonconvex MINLPs in a distributed-memory setting. **?** provide a general framework for parallel branch-and-bound, called Parallel Enumeration and Branch-and-Bound Library (PEBBL (**?**)) that is extended to global optimization of nonconvex NLPs and MINLPs. Parallel versions of NLP-BB (**?**) and OA (**?**) are also available in some MINLP solvers, with the latter exploiting parallelism in the MILP solvers to solve the relaxations.

In this thesis, we study and develop shared-memory parallel extensions of multiple algorithms for MINLP, within a common and open-source MINLP framework, Minotaur. There are no other readily available software that provide all these together. We also study anomalies in these algorithms, not done before in the context of MINLP to the best of our knowledge. We also develop and test novel approaches to solve nonconvex MINLPs (heuristically) and convex MIDFO. The contributions in this thesis are organized into four chapters.

In Chapter 2, we present parallel versions of the existing sequential algorithms for convex MINLP. In particular,

- we develop shared-memory parallel versions of three algorithms for convex MINLP: (i) NLP-BB, (ii) two variants of QG (iii) MILP based outer-approximation (OA), within the Minotaur toolkit.
- we study the effects of different algorithmic components: sharing of information like branching scores amongst different threads, and scalability with the number of threads, on the performance of these algorithms in terms of solution time and tree size.

Parallel approaches could sometimes generate useless (data) tasks, and spend time in (analyzing) executing them. For example, a parallel branch-and-bound algorithm could generate nodes that do not yield a feasible solution and spend a large fraction of time in processing them. For example, a parallel algorithm might choose to solve a node (say $n_1$) and create its children which otherwise could have been pruned by solving another node (say $n_2$) first. This could

result in the parallel algorithm being slower than its sequential counterpart (if it finds an optimal solution earlier). This phenomenon occurs often with parallel extensions of (sequential base) algorithms due to opportunistic (or greedy) or 'ambiguous' decisions within the algorithm. This means that the parallel algorithm may have been designed to deviate too much from its sequential counterpart, yielding unpredictable performance. Such varied behaviour of a parallel algorithm is called an 'anomaly' (**?**). Eliminating anomalies that might worsen the performance of a parallel algorithm and allowing those that enhance their speed are therefore important.

Chapter 3 studies these anomalies in parallel MINLP algorithms. Depending upon how the parallelism is exploited, tree-search algorithms many a times exhibit anomalies, which includes degradation or disproportionate acceleration in running times. This is mainly due to 'ambiguity' of different algorithmic components and we present some unambiguous components in Chapter 3. Overall, on this front,

- we present a parallelization mechanism that solves nodes of the branch-and-bound/cut tree simultaneously in an opportunistic way. Since each processor continuously keeps solving nodes until all the nodes are processed without much synchronization with the other processors, this scheme does not allow reproducibility of runs.

- we implement unambiguous node selection and branching schemes and demonstrate through our experiments that these schemes guarantee no degradation in performance when more than one processors are used in parallel.

- we achieve similar guarantees when an unambiguous cut generation scheme is deployed (in addition to the above) in an LP/NLP based branch-and-bound framework and compare it with the opportunistic version.

- our deterministic algorithms also preserve reproducibility of runs, a highly desirable feature while developing parallel algorithms.

Next, we address nonconvex MINLPs using a parallel branch-and-estimate heuristic. Exact methods for nonconvex MINLP tend to require significant computational time. We focus on obtaining good quality solutions for nonconvex MINLPs within reasonable time. We present a multi-start heuristic to obtain good solutions to nonconvex NLP relaxations by using local NLP solvers and

different initial points. To accelerate the algorithm, we incorporate parallel NLP solving. Multi-start heuristics for continuous NLPs have been proposed earlier (**????**), however, our study focusses on exploiting parallelism in a branch-and-bound framework in the context of MINLP. Chapter 4 presents this heuristic for nonconvex MINLP. In particular,

- we propose a branch-and-bound based heuristic for nonconvex MINLP, where each node in the branch-and-bound tree is a nonconvex NLP and multiple processors are deployed simultaneously to approximately solve each node.
- we deploy a multi-start method to find solutions and bounds to each nonconvex NLP (node) using different sets of initial points.
- we present five different randomized schemes for selecting the initial point.

The work presented in Chapter 5 is an amalgamation of insights from mixed-integer programming and derivative-free optimization (MIDFO). We study MIDFO problems, with integer variables and with an 'integer-convex' objective function (we formally define this notion in Chapter 5). Although a proof of convexity of such a function that has no compact mathematical form or whose derivatives are unavailable is difficult to imagine, however, our assumption rests on the knowledge/physics of the overall system. One of the applications involves optimal design of concentrating solar power plants where one intends to find the number and location of panels on the power plant receiver (**?**). This is a derivative-free setting because each function evaluation requires a complex, computationally expensive simulation (mathematical description of the function and its derivatives is unavailable). Also, some decision variables are integer constrained, and 'unrelaxable', which means that the objective function can not be evaluated at noninteger points. Another application is tuning of codes on high-performance computers, where certain decisions correspond to a set of integer values (**?**), and can not take noninteger values. The optimal material design problem presented by **?** also involves unrelaxable integer constraints. We model and address this difficult class of problems using an algorithm very similar to outer-approximation for MINLPs, although by using only the first-order information on the objective function, that is, only the function values (and no derivative information). In particular,

- we propose a nonconvex piecewise linear underestimator MILP model of the objective function over the feasible region, and prove that it provides a valid lower bound on the optimal objective value.
- we propose an algorithm (similar to outer-approximation) for optimizing such a computationally expensive objective function over a bounded integer lattice and provide a proof that our method converges to a global optimum.
- we propose two computational approaches to construct and tighten the underestimator. In the first approach, we solve a sequence of MILPs. In the second approach, we maintain the values of the lower bounds at the lattice points. Upper bounds on the optimal objective value are obtained using function evaluations and are used to tighten the underestimator
- we present computational results to benchmark our algorithm with state-of-art DFO solvers that can handle integer variables
- we seek insights into why this class of problems is difficult, even for moderate instance sizes, with emphasis on the proof of optimality.

Finally, we present the conclusions and a few research directions based on this thesis in Chapter 6.

# Chapter 2

# Parallel Algorithms for Convex MINLP

This chapter discusses shared-memory parallel implementation of three classical algorithms for convex MINLP: (i) NLP based branch-and-bound, (ii) two variants of LP/NLP based branch-and-bound and (iii) Outer-approximation. We study (a) the effects of different algorithmic components such as sharing of information like branching scores among different threads, and (b) scalability with the number of threads. The proposed parallel algorithms have been implemented within the open-source Minotaur framework (**?**) and tested on benchmark instances from MINLPLib (**?**).

## 2.1 Background

Algorithms for convex MINLP have been implemented in several convex MINLP solvers including AIMMS (**?**), BONMIN (**?**), FilMINT (**?**), Muriqui (**?**), and SHOT (**?**). Global solvers like Antigone (**?**), BARON (**?**), Couenne (**?**), LINDO (**?**) and SCIP (**?**) can also be used to solve convex MINLPs. Global solvers implement heuristics to detect convexity automatically and resort to slower methods for non-convex problems if they fail to detect it. All the stated solvers except SCIP rely on a separate MILP solver for implementing branch-and-cut and related routines. The open-source Minotaur toolkit (**?**) is used to implement the methods proposed in this chapter. Minotaur includes two solvers for convex MINLP: NLP-BB and QG against which we compare the effects of the proposed schemes. While, Minotaur implements its own branch-and-bound tree, it also has the ability to interface

with MILP solvers to use their implementation of tree-search. The latter is used to implement OA and a variant of QG.

Use of shared-memory parallel computing for MILPs has received attention recently, see for example **???**. Most open-source (**??**) and proprietary MILP solvers (**?????**) exploit multiple processors for branch-and-bound/cut framework. Some of the frameworks that exploit shared-memory parallelization are Ubiquity Generator (UG) (**??**), ChiPPS (**?**) and PEBBL (**?**). The UG framework has been used as a parallelization wrapper over many MILP base solvers (**?????**). It explicitly controls the base solver as a callable library by parallelizing the tree-search from outside. FiberSCIP (FSCIP) is the shared-memory parallel algorithm that uses SCIP underneath UG. The frameworks ChiPPS and PEBBL use a master-hub-worker and a hub-worker hierarchy, respectively. The MILP solver **?** implements a multithreaded scheme to parallelize its sequential solver. Nodes are assigned by a master thread to workers sequentially as some of the global data is stored centrally. It also has a deterministic parallelization mode which distributes subtrees to workers instead of nodes. Proprietary software like CPLEX and GUROBI provide LP solvers that can be used as subroutines for solving MINLPs. They also provide MILP solvers that can run in a parallel mode. CPLEX LP and MILP solvers are extensively used in our computational experiments.

## 2.2 Experimental Setup

We have carried out our computational experiments on a server with two 64-bit Intel(R) Xeon(R) E5-2670 v2, 2.50GHz CPUs with 10 cores each and sharing 128GB RAM. Hyperthreading is disabled. Our schemes are implemented in the MINLP toolkit Minotaur[1]. All codes are written in C++, and complied with GCC-4.9.2 compiler. For compiling parallel algorithms, OpenMP-4.0 provided by GCC is used. IPOPT-3.12 with MA97 linear-systems solver is deployed for solving NLPs. For solving the LPs and the MILPs, we use CPLEX-12.8. For this study, out of 374 convex instances in MINLPLib (**?**), we excluded 40 instances that neither have any nonlinearity (in constraints and objective) nor any integer variables after presolving in Minotaur. We used the remaining 334 instances and refer to them as the $TS$ test set in our experiments. Description of these instances and the list of excluded instances can be found in Appendix A of **?**. The wall clock time limit

[1]Available at http://github.com/minotaur-solver/minotaur

is set to one hour for all our experiments and we report all the solution times in seconds.

## 2.3 Shared-Memory Parallel Search

We deploy a parallel tree-search algorithm for solving different nodes of the branch-and-bound tree concurrently using different processors that share a common memory. All 'open' subproblems (associated with nodes) of the branch-and-bound tree, those that are yet to be solved, are stored in a collection called the node-pool. Different nodes are solved in parallel using the *fork-join* model, a commonly used multiprocessing model in shared-memory architectures. The main program is run as a single process which creates multiple 'threads' (**??**) depending on the number of CPUs available and user settings. Threads are capable of doing mutually independent computations like processing different nodes concurrently.

The fork-join model can be thought of as an alternating sequence of forks where various tasks are performed concurrently by multiple threads, and joins, where a single thread performs some serial tasks and synchronization for sharing information between the threads. In our implementation, the main process first reads the MINLP instance, performs some preprocessing and sets up the environment and other required data structures. The main process also creates the threads and starts branch-and-bound. Branch-and-bound then proceeds in rounds. One by one, every thread selects an open node and removes it from the node-pool. Only one thread is allowed to access the node-pool at a time and other threads wait for their turn. This access is on first-come-first serve basis. If there are no nodes available for a thread, it waits until the next round of assignment. Once this selection process is completed, all threads concurrently start solving their respective nodes. When all the threads finish solving their respective nodes, a new round of assignment of open nodes and solving is executed. This process continues until all the open nodes are either processed or pruned and the node-pool becomes empty. We use this fork-join node-level parallelism for the two algorithms: NLP-BB and QG.

We have implemented our fork-join model using the OpenMP directives (see **?**). OpenMP directives provide a simple way of specifying concurrency, synchronization and data handling - without the need to explicitly create threads, allocate memory, delete memory, etc. While this approach provides fewer features

and lesser flexibility than POSIX threads (popularly called Pthreads) or standard threads provided by C++11, it simplifies multithreaded programming to a great extent. Unlike Pthreads, OpenMP provides higher level constructs called 'directives' which can be used directly without specifying thread level details. An OpenMP directive has the following form.

```
#pragma omp <directive> [clause list]
```

The beginning of a parallel region in the code is marked by a directive called `parallel`. We use the following common clauses with the `parallel` directive in our algorithms.

- `for`: This clause is for assigning tasks (individual iterations of the for loop) to different threads. These tasks (for example, create or update a relaxation, solve a relaxation, etc.) are executed in parallel.
- `critical (<name>)`: This clause is used for synchronization, which prohibits two or more threads from executing blocks of the same `name`.
- `single`: This clause is used within parallel regions where execution by any one thread suffices (for example, to check stopping conditions).
- `omp_set_num_threads(<natural number>)`: This clause is used for specifying the degree of concurrence (the number of desired parallel threads). It is typically set equal to the number of available processors.

An in-depth description of OpenMP programming can be found in the book by **?**.

While a more detailed description of Minotaur design and its C++ classes is available in **?**, we briefly describe the important classes that are used in our parallel implementation. The program starts by reading the problem, and then *presolves* it using the `Presolver` class. The presolved problem is then passed to the `NodeRelaxer` class which creates a relaxation. A node is processed using the `NodeProcessor` class, that deploys an appropriate LP or NLP solver called through the `LPEngine` or `NLPEngine` class, respectively. If an optimal solution of the relaxation is found, and if this solution is not feasible to the MINLP, the `NodeProcessor` calls a `Brancher` class to find a suitable branching candidate. The class `TreeManager` handles all the tree-related information like nodes, upper and lower bounds, etc. Using the branches found by the `Brancher`, two new child nodes are created by the `TreeManager`.

We preserve the basic design of the sequential branch-and-bound in Minotaur and utilize the existing classes, which makes our implementation lightweight. As in the serial version, we maintain a single, central `TreeManager` which stores and maintains all node descriptions. Each thread individually maintains a private copy of all the necessary class objects, like `NodeRelaxer`, `NodeProcessor`, `Brancher`, etc., and acts as an independent unit that synchronizes with other threads at the end of each round. The first thread starts solving the root relaxation while the other threads wait. If branching is required, the thread creates two child nodes. In the next round of node selection, one of the other idle threads obtains a node. Each thread that has a node now processes its respective node in the next round and the process continues. When sufficient number of open nodes are available, all threads become busy. If $T$ number of threads are used, the ramp-up time before all threads are busy is at least $\lceil log_2(T) \rceil$ times the average node solving time. When the node-selection strategy is based on diving (**?**), each thread retains one of the children of the node it solved in the previous round for quick warm-starting of LPs or NLPs. Each thread maintains a private copy of the original MINLP to create relaxations of the nodes that it receives and to check whether a relaxation yields a feasible solution to the MINLP. After each round of solving, stopping conditions are checked by any one of the threads. The search terminates when all open nodes are exhausted (solved, pruned by bound or pruned by infeasibility) or some other stopping condition (time limit, node limit, etc.) is met. The schematics of the parallel tree-search and the *Process* block are shown in Figure 2.1. *Process* block refers to the sets of tasks involved in processing a subproblem.

### 2.3.1 Parallel Extension of NLP-BB

The scheme shown in Figure 2.1 can be viewed as the parallel NLP-BB algorithm, where the nodes in the tree are NLP relaxations and an `NLPEngine` (NLP subroutine) is used to solve them. We denote this parallel solver in Minotaur as *mcbnb* and study its performance when using different number of threads. The hardware and software setup mentioned in Section 2.2 has been used in these experiments as well. The NLP solver IPOPT (**?**) (version 3.12) with MA97 linear-systems solver is thread-safe, hence suitable for our parallel algorithm.

The scalability of our implementation with the number of threads is depicted by what we call 'Scalability Graph'. We use Shifted Geometric Mean (SGM) for reporting the performance measures. Due to its robustness, SGM is often used in

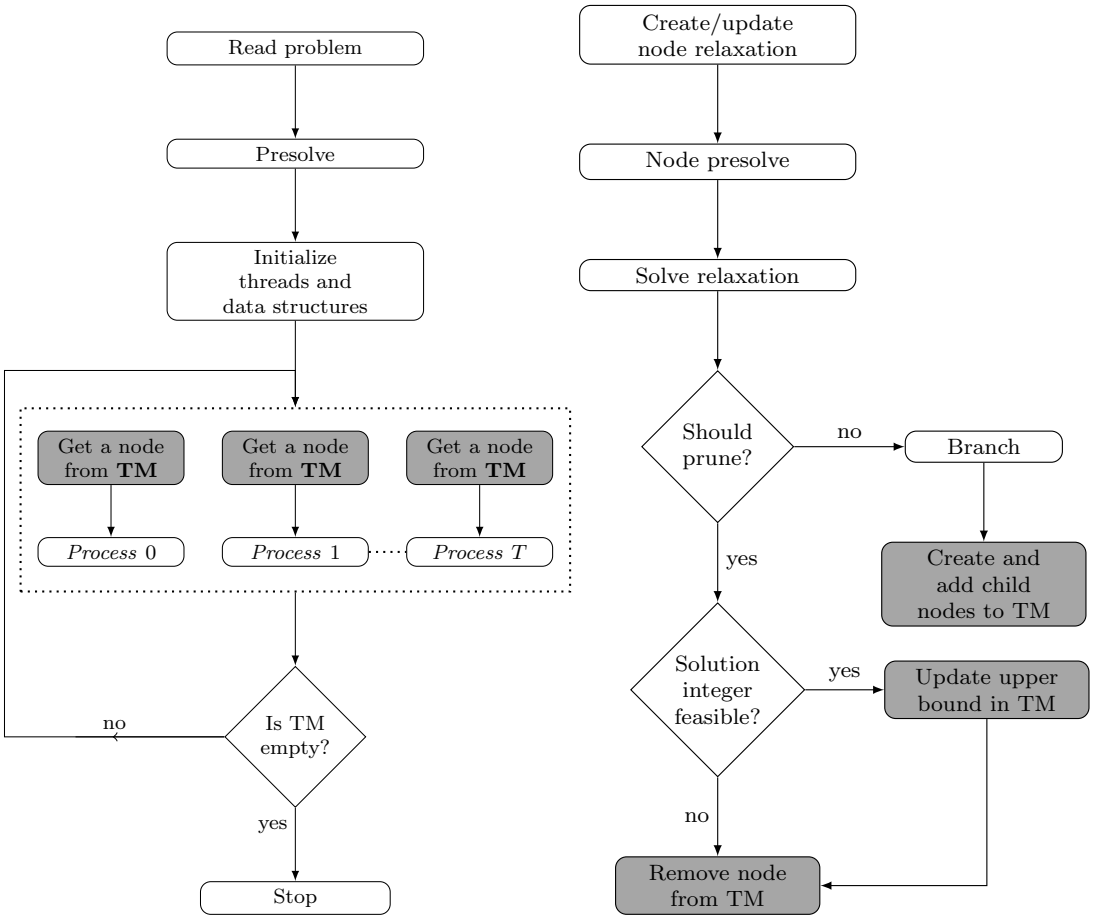



**Figure 2.1:** Schematics of the parallel tree-search (left) in Minotaur and the *Process* block (right). Gray-colored blocks involving the `TreeManager` (denoted **TM**) are critical. The block where stopping condition is checked is executed by any one of the threads.

computational studies (see for example, **?**). While SGM gives the mean improvement over all instances, this graph shows the distribution of performance over the test set. It is a line plot with each line corresponding to an algorithm with fixed thread-count. Each line plots the fraction of instances that can be solved within a $w$-factor of time taken by the single-thread run. Given a set of instances, $\mathcal{TS}$, the graph is plotted as a non-decreasing line graph. For each value $w$, it plots

$$\frac{\left|\{i \in \mathcal{TS} : t_{i,T} \leq w t_{i,1}\}\right|}{|\mathcal{TS}|},$$

where $t_{i,T}$ is the time taken by the solver when running $T$ threads on instance $i$. If the solver does not finish solving within the time limit, $t_{i,T}$ is set to infinity. The ratios we use are different from the ones used in performance profiles by **?**, where the ratios are calculated with respect to the time taken by the fastest solver for each instance.

Figure 2.2 shows the scalability graphs for *mcbnb*. The X-axis represents the factor $w$. The plot for *mcbnb1* (*mcbnb* with one thread), the reference solver, is a step function by definition. Its height (about 0.7 in this case) is the fraction of instances that could be solved within the time limit by the single-thread run. The plot for *mcbnb2* shows that it could solve about 5% (the value at $2^{-1}$) of the instances faster by a factor of two or more as compared to *mcbnb1*. Similarly, *mcbnb4* and *mcbnb16* could solve about 20% and 30% respectively for the same. The rightmost value on the plot shows the fraction of instances that could be solved within the time limit.

SGM values for wall clock time and nodes processed are reported in Table 2.1. The first column ('# threads (T)') in the top table in Table 2.1 indicates the number of threads used. A 'T' at the end of the solver name indicates the number of threads used by it. Also, 'wall time' denotes the wall clock time (not the CPU time) taken by the multithreaded code. Each row of the top table in Table 2.1 corresponds to the solver with T threads. The column '# solved by' lists the number of instances solved by the corresponding solver and by both the reference solver (*mcbnb1* in this case) as well as the multithreaded solver (under the column 'both'). The first column under the headings 'wall time' and 'nodes' shows the shifted geometric mean (SGM) of these measures reported by the reference solver for the instances in the column 'both'. The second column under these headings shows the relative SGM ('rel.') of the proposed method for the same set of instances. We used a shift of 10 for calculating SGM of time and 100 for the number of nodes processed.

Using 16 threads, *mcbnb* could solve 17 additional instances compared to *mcbnb1*, and achieved a speed-up of about two on average. The growth in tree-size with increasing number of threads is well below linear, which ultimately leads to gains in parallelism. The bottom table in Table 2.1 shows the statistics for the best solver (*mcbnb16* in this case) when instances are categorized based on difficulty level. The improvements due to parallelism are more prominent for difficult instances (row corresponding to time > 100).

### 2.3.2 Sharing Pseudocosts in Branching

The implementations of NLP-BB and QG algorithms in Minotaur use the well known reliability branching scheme by **?**. Reliability branching uses strong branching (**??**) initially to find the score of branching candidates. As strong branching is expensive, the scheme uses previously computed scores after a cer-

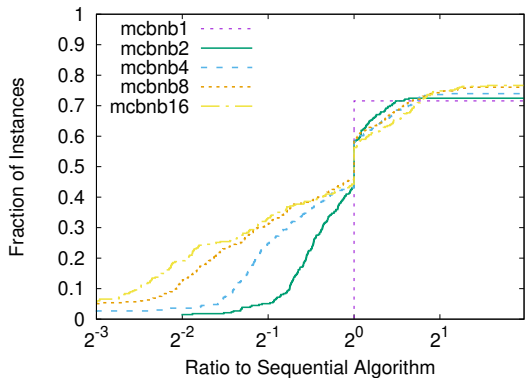



**Figure 2.2:** Scalability graphs of wall clock times taken by *mcbnb* using different number of threads on test set $TS$.

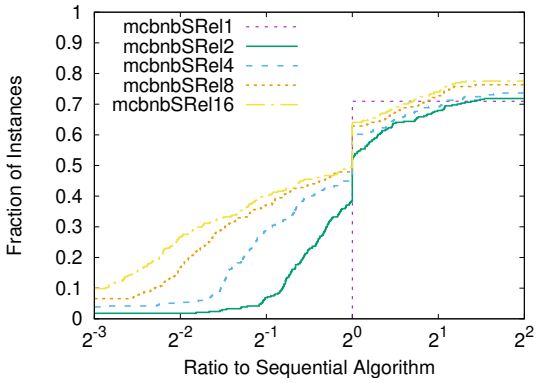



**Figure 2.3:** Scalability graphs of wall clock times taken by multithreaded variants of *mcbnbSRel* on test set $TS$.

**Table 2.1:** (Top) Comparison of *mcbnb1* with *mcbnb* using multiple threads on test set $TS$. *mcbnb1* could solve 239 instances. (Bottom) Break-up of performance of *mcbnb16* over instances of varying difficulty.

| # threads | # solved by | | wall time | | nodes | |
|---|---|---|---|---|---|---|
| (T) | *mcbnbT* | both | *mcbnb1* | rel. | *mcbnb1* | rel. |
| 2 | 242 | 238 | 32.73 | 0.80 | 3.1e2 | 1.05 |
| 4 | 247 | 239 | 33.53 | 0.69 | 3.2e2 | 1.16 |
| 8 | 254 | 239 | 33.53 | 0.60 | 3.2e2 | 1.28 |
| 16 | 256 | 239 | 33.53 | 0.56 | 3.2e2 | 1.54 |

| | # solved | wall time | | nodes | |
|---|---|---|---|---|---|
| time | by both | *mcbnb1* | rel. | *mcbnb1* | rel. |
| > 0 | 239 | 33.53 | 0.56 | 3.2e2 | 1.54 |
| > 10 | 132 | 113.58 | 0.41 | 9.6e2 | 1.57 |
| > 100 | 63 | 428.05 | 0.28 | 2.6e3 | 1.40 |
| > 500 | 25 | 1153.12 | 0.30 | 5.3e3 | 1.60 |

tain number of strong-branching trials. In a parallel setting, the scores obtained at a node by a thread may be useful at nodes processed by other threads. However, sharing this information comes at the cost of querying additional information (from other threads), which means that each thread has to spend additional time in gathering and processing this information.

We implemented reliability branching for a parallel setting in two different ways. In the first way which we call *privateRel*, each thread does reliability branching independent of other threads using information from only the nodes

it has processed earlier. In the second way which is referred to as *sharedRel*, each thread uses information from the nodes solved by other threads also. This aspect is illustrated in Figure 2.4. Suppose, for instance, we have two threads, then the first thread, `thread0`, solves the root node indexed 0 in the first round and then one red-colored node in each subsequent round. Simultaneously, the other thread, `thread1`, starts solving the hatched nodes, starting from the node indexed 2. In *privateRel*, both `thread0` and `thread1` use the information generated only at the nodes they solve. The other brancher, *sharedRel*, queries the node-solve information from the other threads at the end of each round and uses the cumulative information (from yellow-colored, red-colored and hatched nodes) to decide the branching variable at a node. The accumulation of information like pseudocosts (**?**), number of times branched, etc., from other threads to calculate scores requires an additional query by each thread. However, these queries turn out to be beneficial overall as they do not consume much time and are typically executed in parallel, independently at each thread.

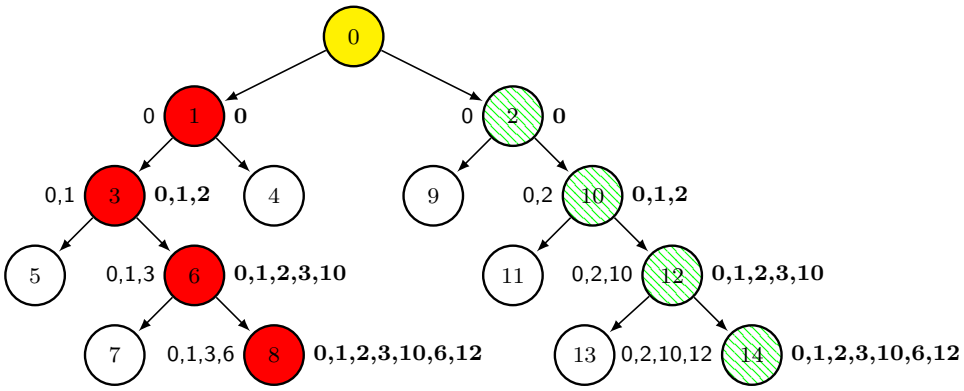



**Figure 2.4:** Illustration of using pseudocosts by two threads for branching. The root node indexed 0 and then the red-colored nodes are solved by `thread0` and the hatched nodes are solved by `thread1`. In *privateRel*, `thread0` uses pseudocosts only from the yellow and the red-colored nodes while `thread1` uses pseudocosts from only the yellow-colored and the hatched nodes (indices shown on the left of each node). In *sharedRel*, information from all the processed nodes is used by both the threads (indices shown on the right of each node).

Figure 2.3 shows the effect of sharing pseudocosts in *mcbnb* when using multiple threads. This version is referred to as *mcbnbSRel*. We see that sharing pseudocosts after each round is beneficial, and the benefits grow with the number of threads. As shown in Table 2.2, sharing pseudocosts enabled *mcbnbSRel16* to solve 4 more instances than *mcbnb16* (21 more compared to *mcbnb1*). Also,

the mean wall clock time is reduced to a fourth for difficult instances (row corresponding to time > 500) using *mcbnbSRel16*.

**Table 2.2:** (Top) Comparison of *mcbnbSRel1* with *mcbnbSRel* using multiple threads on test set $TS$. *mcbnbSRel1* could solve 237 instances. (Bottom) Break-up of performance of *mcbnbSRel16* over instances of varying difficulty.

| # threads | # solved by | | wall time | | nodes | |
|---|---|---|---|---|---|---|
| (T) | *mcbnbSRelT* | both | *mcbnbSRel1* | rel. | *mcbnbSRel1* | rel. |
| 2 | 241 | 236 | 30.71 | 0.86 | 3.0e2 | 1.11 |
| 4 | 247 | 236 | 30.73 | 0.69 | 3.0e2 | 1.24 |
| 8 | 255 | 237 | 30.23 | 0.56 | 3.1e2 | 1.40 |
| 16 | 260 | 237 | 30.23 | 0.50 | 3.1e2 | 1.59 |

| | # solved | wall time | | nodes | |
|---|---|---|---|---|---|
| time | by both | *mcbnbSRel1* | rel. | *mcbnbSRel1* | rel. |
| > 0 | 237 | 31.23 | 0.50 | 3.1e2 | 1.59 |
| > 10 | 130 | 104.24 | 0.37 | 9.0e2 | 1.64 |
| > 100 | 63 | 364.46 | 0.27 | 2.1e3 | 1.61 |
| > 500 | 23 | 1008.44 | 0.25 | 4.8e3 | 1.65 |

### 2.3.3 Parallel Extension of QG

The implementation of parallel QG algorithm in Minotaur differs from *mcbnb* in two ways. First, an LP solver is used to solve the (LP) relaxation at each node. Second is the generation and sharing of globally valid linearization cuts that are generated at certain nodes either after solving an NLP or by linearization methods like those described by **?**. Recall that linearization cuts are added in QG when integer feasible points are encountered in the branch-and-bound tree. To strengthen the relaxations at the root node and other nodes of the tree, **?** propose linearization schemes. The first set of schemes tighten the LP relaxation at the root node and the second set of schemes are for adding new linearizations down in the branch-and-bound tree. These strategies are based on change in the lower bound, depth of the nodes, some appropriate measures of constraint violation, problem structure, etc.

In order to store and share these cuts, first we add them to a local `CutPool` of a thread. A `CutManager` class is used by each individual thread to store all the linearizations generated while processing the nodes assigned to it. A thread

queries the `CutManager` of all other threads while creating the relaxation of the node assigned to it, and all cuts that are *new* for this thread are added to this relaxation. The cuts from `CutManagers` of different threads that have been added to the relaxation at a given thread are maintained and updated using a unique cut id. We denote this parallel QG algorithm as *mcqg*. Algorithm 2.1 demonstrates the *mcqg* algorithm implemented within Minotaur, and Algorithm 2.2 describes the function *GetNode*() used in Algorithm 2.1.

**Algorithm 2.1:** Parallel QG (LP/NLP based branch-and-bound) algorithm in Minotaur.

1 Initialize upper bound, $U = \infty$, state of thread, $S_t = idle$, cut pool, $C_t = \emptyset,\ \forall\ t \in 1 \ldots T$.
2 Add root LP relaxation to the pool of open nodes, $\mathcal{H}$.
3 **while** $\mathcal{H} \neq \emptyset$ **do**
4  **for** *t in* $1 \ldots T$ **do**
5  **if** $S_t = idle$ **then**
6  *GetNode()*.
7  **if** $S_t = assigned$ **then**
8  Add *new* cuts from $C_t$, $\forall t$ in $1 \ldots T$, to $LP_t$.
9  Solve $LP_t$ at thread $t$.
10  **if** $LP_t$ *is optimal and* $(\hat{x}^t)_i \in \mathbb{Z},\ \forall\ i \in \mathcal{I}$ **then**
11  Solve F-NLP($\hat{x}^t$), let the point returned be $\check{x}^t$.
12  **if** *F-NLP*($\hat{x}^t$) *is optimal* **then**
13  Update $U \leftarrow \min\{U, f(\check{x}^t)\}$.
14  Generate linearizations of all nonlinear constraints violated by $\hat{x}^t$, at $\check{x}^t$, and add to $C_t$ and $LP_t$.
15  Go to step 9.
16  **else if** $LP_t$ *is infeasible* **then**
17  Prune this node.
18  *GetNode()*.
19  **else**
20  Branch: generate two *LP* subproblems and add to $\mathcal{H}$.
21  *GetNode()*.

**Algorithm 2.2:** Get an open node from $\mathcal{H}$ for a thread $t \in \{1, \ldots, T\}$

```
1 if H ≠ ∅ then
2     Remove an LP from H as per the search strategy and set LP_t ← LP
      and S_t = assigned.
3 else
4     Set S_t = idle.
```

Table 2.3 summarizes the performance of multithreaded variants of *mcqg* relative to *mcqg1*. All threads share linearizations (at integer solutions) and pseudocosts according to *sharedRel* scheme. We observed improvements with all the variants of *mcqg* over *mcqg1*. About 44% improvement in wall clock time is obtained when using 16 threads and 9 more instances were solved. The scalability graphs for *mcqg* are shown in Figure 2.5.

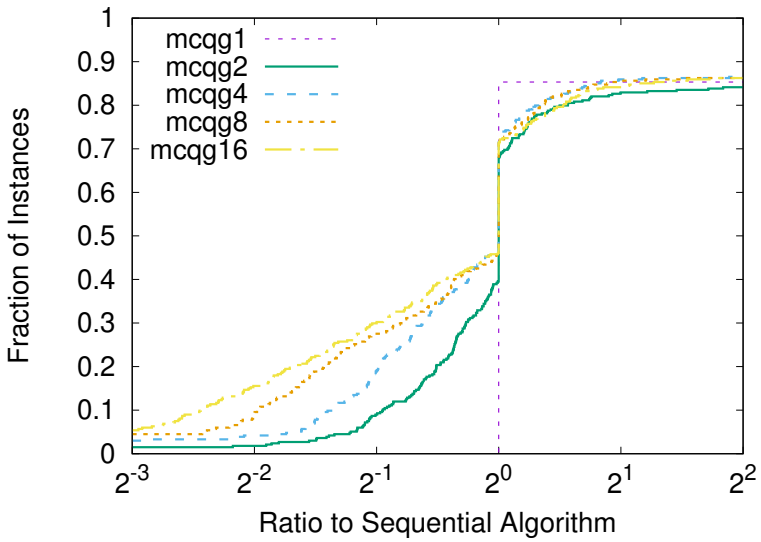



**Figure 2.5:** Scalability graphs of wall clock times for *mcqg* variants on test set $TS$.

## 2.4 Combined Effect of Linearization and Parallelization Schemes

Our numerical experiments show that parallelism in tree-search enhances the performance of *qg* with the hybrid linearization scheme Hyb proposed by **?**. This linearization scheme entails conditions using which additional gradient inequalities can be added at different nodes in the search tree. We refer to the combination of *mcqg* with Hyb as *mcqgHyb*. Table 2.4 shows the performance of *mcqgHyb16*

**Table 2.3:** (Top) Comparison of *mcqg1* to *mcqg* using multiple threads on test set $TS$. *mcqg1* could solve 285 instances. (Bottom) Break-up of performance of *mcqg16* over instances of varying difficulty.

| # threads | # solved by | | wall time | | nodes | |
|---|---|---|---|---|---|---|
| (T) | *mcqgT* | both | *mcqg1* | rel. | *mcqg1* | rel. |
| 2 | 283 | 279 | 19.44 | 0.86 | 2.1e3 | 1.08 |
| 4 | 291 | 282 | 19.50 | 0.75 | 2.1e3 | 1.17 |
| 8 | 291 | 280 | 19.62 | 0.64 | 2.1e3 | 1.20 |
| 16 | 294 | 284 | 20.08 | 0.56 | 2.2e3 | 1.29 |

| | # solved | wall time | | nodes | |
|---|---|---|---|---|---|
| time | by both | *mcqg1* | rel. | *mcqg1* | rel. |
| > 0 | 284 | 20.08 | 0.56 | 2.2e3 | 1.29 |
| > 10 | 120 | 100.52 | 0.39 | 3.3e4 | 1.32 |
| > 100 | 56 | 331.22 | 0.27 | 1.2e5 | 1.30 |
| > 500 | 19 | 1199.84 | 0.24 | 4.1e5 | 1.19 |

(*mcqgHyb* with 16 threads) and *qgHyb* (*qg* with Hyb) in comparison to *qg* on test set $TS$. Note that the wall clock time taken by the sequential algorithm *qg* is the same as the CPU time. Using *mcqgHyb16* on $TS$, we observed a significant improvement of about 44% in the solution times and solved 6 instances more than *qg* and 5 more than *qgHyb*.

**Table 2.4:** (Top) Comparison of *qgHyb* and *mcqgHyb16* to *qg* on test set $TS$. *qg* could solve 291 instances. (Bottom) Break-up of results of *mcqgHyb16* over instances of varying difficulty.

| method | # solved by | | wall time | | nodes | |
|---|---|---|---|---|---|---|
| (M) | M | both | *qg* | rel. | *qg* | rel. |
| *qgHyb* | 292 | 288 | 18.09 | 0.88 | 2.2e3 | 0.83 |
| *mcqgHyb16* | 297 | 288 | 18.09 | 0.56 | 2.2e3 | 1.14 |

| | # solved | wall time | | nodes | |
|---|---|---|---|---|---|
| time | by both | *qg* | rel. | *qg* | rel. |
| > 0 | 288 | 18.09 | 0.56 | 2247.04 | 1.14 |
| > 10 | 117 | 93.35 | 0.40 | 38730.08 | 1.08 |
| > 100 | 45 | 485.08 | 0.29 | 204398.59 | 0.92 |
| > 500 | 22 | 1178.97 | 0.30 | 474288.37 | 1.07 |

## 2.5 Outer-Approximation with Parallelism in MILP

As briefly explained in Section 2.1, the underlying strategy in outer-approximation based algorithms is to solve an alternating sequence of MILPs (of the form (RM)) and NLPs (of the form F-NLP). This section describes two variants of OA that exploit parallelism of the MILP solver.

### 2.5.1 Multitree OA with Parallel MILP

As OA is an iterative scheme in which an MILP and a fixed-NLP are solved alternatingly, a natural way of parallelizing it is to use a parallel MILP solver. We have implemented the default OA scheme in Minotaur and also enhanced it in the following way. We solve MILP relaxation at any iteration using an MILP solver. The MILP solver can utilize all the available processors. MILP solvers also have the capability of returning a pool of solutions which we use to generate additional linearizations. For each solution $x^t$ in the pool returned by the MILP solver, we solve the corresponding fixed-NLP F-NLP($x^t$) and generate the linearizations. These NLPs can in turn be solved in parallel if the NLP solver is thread-safe. All the generated linearizations are added to the MILP. When all NLPs have been solved and linearizations added, the MILP solver is called again and the process continues. Algorithm 2.3 describes the steps of the enhanced OA. We also solve multiple F-NLPs, each corresponding to a distinct MILP solution, in parallel (indicated by the `for` loop in Algorithm 2.3).

In order to further accelerate the MILP solver, we use the MIP starts functionality (also called advanced starts or warm starts) provided by the MILP solver, CPLEX in our case. The solutions obtained by it are written to a file and are read in the subsequent MILP call. In our experiments, we observed that CPLEX was able to repair some of the solutions from the MIP starts and obtain upper bounds, mainly because the MILPs in consecutive iterations differ only by a few linear constraints. Additionally, we provide the best known upper bound of (P) to the MILP solver in each iteration to be used as a cut-off value. In Minotaur, we interact with the CPLEX solver using a C++ wrapper that passes information to and from CPLEX through its C interface.

We compare the performance of our two implementations of multitree OA. In the first implementation, linearizations are added only at the point obtained from the optimal solution of MILP. The second one uses all solutions of the solution pool of MILP, and solves fixed-NLPs in parallel using multiple threads.

**Algorithm 2.3:** Exploiting solution pool of MILP solver in multitree OA

1 Initialize bounds, $U = \infty, L = -\infty$, iteration counter $k = 0$.
2 Solve the NLP relaxation (R). If (R) is infeasible, then so is (P) and we STOP. If the optimal solution of (R), $x^0$, is feasible for (P), then set $U = f(x^0) = L$ and STOP.
3 Create and solve the MILP relaxation (RM). If (RM) is infeasible, then so is (P) and we STOP. Otherwise, let $\hat{X}^k$ be the set of available feasible solutions of (RM), $\hat{z}^k$ be its optimal value, and set $L = \hat{z}^k$.
4 **while** $U > L$ **do**
5  **for** $x^t \in \hat{X}^k$ **do**
6  Solve F-NLP($x^t$), let the point returned be $\check{x}^t$. If F-NLP($x^t$) is optimal, update $U \leftarrow \min\{U, f(\check{x}^t)\}$.
7  Add linearizations to nonlinear constraints violated by $x^t$, at $\check{x}^t$, to (RM).
8  Set $k \leftarrow k + 1$, solve (RM), and update $L \leftarrow \hat{z}^k$.

We denote these implementations of OA as *oa* and *oaSol*, respectively. Table 2.5 and Table 2.6 provide a summary of performance of these algorithms. Here, we present the SGM for the number of iterations taken by *oa*. We observe that the use of solution pool enables us to solve more instances. One can also solve fixed-NLPs one by one if a thread-safe NLP solver is not available. In our experiments, we find that using the solution pool and solving fixed-NLPs in parallel is the most effective strategy. Compared to the traditional OA (*oa1*), we could solve up to 13 more instances and improve the wall clock time by more than 50%.

### 2.5.2 QG Using MILP Solvers with Lazy Cuts Callback

This version of QG is also known as the *Single-tree* OA because it explores a single tree, but uses an MILP solver for creating the tree. MILP solvers like CPLEX and GUROBI provide the users with callback functions which can be invoked in specific *contexts*, for example, when an integer feasible solution is found in the MILP tree. In such contexts, the MILP solving is paused and the control is transferred (temporarily) to a predeclared user-callback function. The user can access MILP solving information, for example, the best solution, upper and lower bounds, etc., generated within the MILP solver so far. This information can then

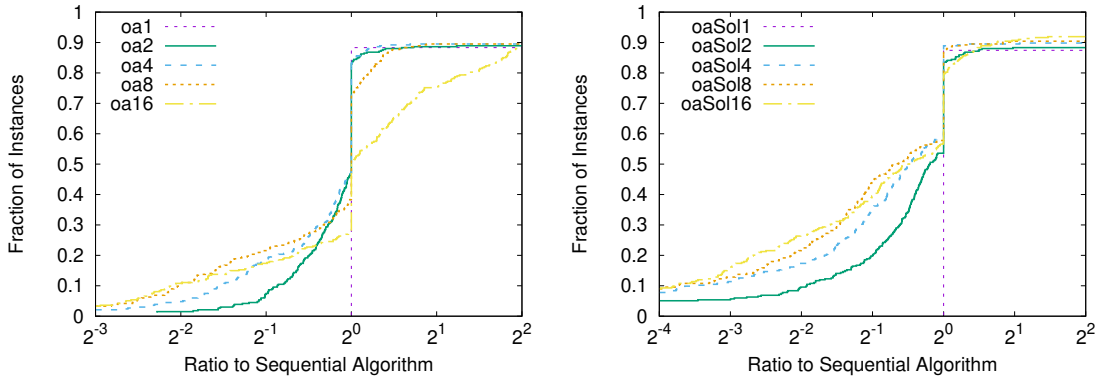



**Figure 2.6:** (Left) Effect of providing multiple threads to CPLEX in *oa* on test set *TS*. (Right) Performance of *oaSol* that uses the solution pool of CPLEX and solves fixed-NLPs in parallel.

**Table 2.5:** (Top) Comparison of *oa* using multiple threads. *oa1* could solve 296 instances. (Bottom) Break-up of *oa16* results over instances of varying difficulty.

| # threads | # solved by | | wall time | | iterations | |
|---|---|---|---|---|---|---|
| (T) | *oaT* | both | *oa1* | rel. | *oa1* | rel. |
| 2 | 299 | 295 | 11.44 | 0.80 | 12.44 | 1.00 |
| 4 | 301 | 295 | 11.44 | 0.68 | 12.44 | 1.01 |
| 8 | 302 | 295 | 11.44 | 0.62 | 12.44 | 1.01 |
| 16 | 302 | 295 | 11.44 | 0.84 | 12.44 | 1.01 |

| | # solved | wall time | | iterations | |
|---|---|---|---|---|---|
| time | by both | *oa1* | rel. | *oa1* | rel. |
| > 0 | 295 | 11.44 | 0.84 | 12.44 | 1.01 |
| > 10 | 109 | 54.18 | 0.67 | 37.47 | 1.02 |
| > 100 | 36 | 335.70 | 0.31 | 48.23 | 0.98 |
| > 500 | 12 | 1161.57 | 0.22 | 73.49 | 0.94 |

be utilized in the callback to generate new cuts, feasible solutions, etc. that are passed back to the MILP solver through predefined functions. When solving convex MINLPs, the MILP solver is not aware of the nonlinear constraints. When an integer feasible solution to the MILP is obtained, it has to be checked for nonlinear constraints. If the obtained solution violates any of them, linearization cuts generated using this point are added to the MILP as 'lazy' cuts, which cut this solution off. In this way, the MILP tree is guided towards an optimal solution of the original problem (P). In this algorithm, the MILP solver maintains the MILP

**Table 2.6:** (Top) Comparison of *oaSol* using multiple threads. *oaSol1* could solve 290 instances. (Bottom) Results of *oaSol16* over instances of varying difficulty.

| # threads | # solved by | | wall time | | iterations | |
|---|---|---|---|---|---|---|
| (T) | *oaSolT* | both | *oaSol1* | rel. | *oaSol1* | rel. |
| 2 | 297 | 288 | 13.92 | 0.63 | 16.75 | 0.66 |
| 4 | 302 | 288 | 13.81 | 0.50 | 16.79 | 0.54 |
| 8 | 304 | 290 | 14.17 | 0.45 | 16.52 | 0.60 |
| 16 | 309 | 290 | 13.94 | 0.43 | 16.69 | 0.58 |

| | # solved | wall time | | iterations | |
|---|---|---|---|---|---|
| time | by both | *oaSol1* | rel. | *oaSol1* | rel. |
| > 0 | 290 | 13.94 | 0.43 | 16.69 | 0.58 |
| > 10 | 108 | 67.46 | 0.27 | 40.24 | 0.51 |
| > 100 | 35 | 401.20 | 0.16 | 62.52 | 0.58 |
| > 500 | 14 | 1255.97 | 0.09 | 98.26 | 0.36 |

tree, along with most of its advanced MILP solving features like presolving, implications, heuristics, etc., that help accelerate the overall tree-search.

This implementation is similar to the multitree OA. First, the root MILP relaxation is passed to the MILP solver. Before solving the MILP, we activate the lazy constraints callback function in the MILP solver. Whenever the MILP solver finds an integer feasible solution, say $x^t$, it returns the control back to Minotaur through a predefined callback. We solve F-NLP($x^t$) in the callback, generate linearization cuts for all nonlinear constraints active at the solution and pass them to the MILP solver which then resumes the MILP tree-search. All the available processors are utilized by the MILP solver within its algorithm. We observed that CPLEX sets the parallel tree-search mode to *deterministic* when using the lazy cuts callback, and only one thread is allowed to access the callback at a time. We conducted two sets of experiments: one with the *deterministic* mode and the other by explicitly setting the parallel mode of CPLEX to *opportunistic*. The latter mode does not guarantee reproducibility of results, so we performed 5 replications. For each instance in test set $TS$, its solution time is computed as the arithmetic mean of the 5 replications. Table 2.7, Table 2.8 and Figure 2.7 present the performance of `deterministic` (*lstoaD*) and `opportunistic` (*lstoaO*) modes. We observed good scalability with *lstoaO*. Using 16 threads, both solution time and tree-size improved by more than 60%. On the other hand, *lstoaD* did not show scalability,

probably due to the sequential NLP solving. Although, both *qg* and *lstoa* are implementations of QG, use of advanced MILP solving techniques within *lstoa* leads to better performance when compared to *qg*. We discuss it next.

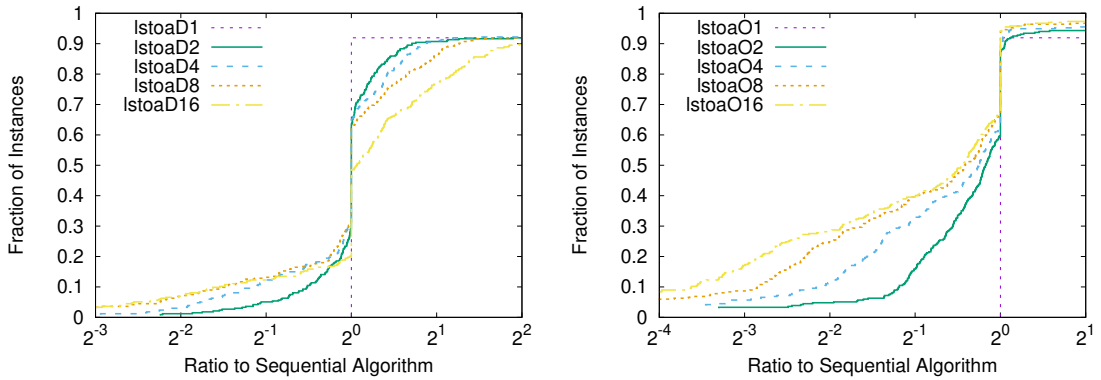



**Figure 2.7:** Effect of providing multiple threads to *lstoaD* and *lstoaO* (*qg* implemented using CPLEX with lazy cuts callback functionality using *deterministic* (left) and *opportunistic* parallel mode) on test set $TS$.

**Table 2.7:** (Top) Comparison of *lstoaD* using multiple threads. *lstoaD1* could solve 307 instances. (Bottom) Break-up of results of *lstoaD16* over instances of varying difficulty.

| # threads | # solved by | | wall time | | nodes | |
|---|---|---|---|---|---|---|
| (T) | *lstoaDT* | both | *lstoaD1* | rel. | *lstoaD1* | rel. |
| 2 | 308 | 306 | 9.86 | 0.93 | 8.6e2 | 1.05 |
| 4 | 309 | 306 | 9.86 | 0.83 | 8.6e2 | 1.07 |
| 8 | 309 | 306 | 9.86 | 0.80 | 8.6e2 | 1.09 |
| 16 | 309 | 306 | 9.86 | 0.92 | 8.6e2 | 1.17 |

| | # solved | wall time | | nodes | |
|---|---|---|---|---|---|
| time | by both | *lstoaD1* | rel. | *lstoaD1* | rel. |
| > 0 | 306 | 9.86 | 0.92 | 8.6e2 | 1.17 |
| > 10 | 111 | 42.74 | 0.79 | 1.2e4 | 1.23 |
| > 100 | 32 | 299.94 | 0.41 | 6.7e4 | 1.01 |
| > 500 | 13 | 871.90 | 0.25 | 1.4e5 | 0.94 |

## 2.6 Comparison of Methods and Conclusions

In the final part of our study, we compare these enhanced routines to each other and also to other MINLP solvers. The goal of this comparison is not to benchmark these solvers, but rather to understand the broad effects of the choice of

**Table 2.8:** (Top) Comparison of *lstoaO* using multiple threads. *lstoaO1* could solve 307 instances. (Bottom) Break-up of results of *lstoaO16* over instances of varying difficulty.

| # threads | # solved by | | wall time | | nodes | |
|---|---|---|---|---|---|---|
| (T) | *lstoaOT* | both | *lstoaO1* | rel. | *lstoaO1* | rel. |
| 2 | 317 | 307 | 12.32 | 0.74 | 8.6e2 | 0.07 |
| 4 | 318 | 305 | 12.35 | 0.57 | 8.6e2 | 0.06 |
| 8 | 323 | 307 | 12.32 | 0.44 | 8.6e2 | 0.08 |
| 16 | 325 | 307 | 12.32 | 0.37 | 8.6e2 | 0.07 |

| | # solved | wall time | | nodes | |
|---|---|---|---|---|---|
| time | by both | *lstoaO1* | rel. | *lstoaO1* | rel. |
| > 0 | 307 | 12.32 | 0.37 | 8.6e2 | 0.07 |
| > 10 | 109 | 57.27 | 0.22 | 1.3e4 | 0.01 |
| > 100 | 28 | 453.24 | 0.10 | 9.3e4 | 0.00 |
| > 500 | 13 | 1024.61 | 0.08 | 8.8e4 | 0.01 |

algorithms and implementation details on their performance. We consider the serial and parallel versions of four algorithms described in this chapter.

1. NLP-BB with sharing of branching information between threads (*mcbnb-SRel*).
2. QG with extra linearizations and parallelization using our own branch-and-cut implementation (*mcqgHyb*).
3. QG with branch-and-cut implementation of CPLEX MILP solver running in opportunistic mode (*lstoaO*).
4. OA with CPLEX MILP solver using all solutions from the solution pool of CPLEX (*oaSol*).

We also include two other MINLP solvers that support parallelization: FSCIP (**?**) and SHOT (**?**). FSCIP is a shared-memory variant of the MILP and MINLP solver SCIP (**?**). SCIP was initially developed for MILPs and was later extended by **?** to global optimization. Developed in C language, it has several plugins that exploit problem structure for branching, presolving, heuristic search, cutting planes, conflict analysis, etc. SCIP can call several LP solvers including CPLEX and also the NLP solver IPOPT for solving relaxations. As mentioned in Section 2.1, FSCIP uses the UG framework to call separate SCIP instances at each thread. Open subproblems are distributed to each thread which then solve the respective subtrees.

UG also dynamically controls and manages the load at each thread. SHOT was developed recently for solving convex MINLPs. It implements ESH and ECP based algorithms (similar to outer-approximation) that solve a sequence of MILP subproblems. SHOT also has a lazy cuts based QG algorithm. SHOT depends on parallelism that the MILP solver exploits in both these algorithms. For our experiments, we use the default ESH and lazy cuts based QG algorithm (also called single-tree polyhedral outer-approximation by SHOT (**?**)).

We compiled SCIP, SHOT and Minotaur using the same versions of CPLEX (LP and MILP) and IPOPT (NLP) subsolvers. Also, we maintained all the default settings of these solvers except in FSCIP, where we disabled convexity detection routines by setting `constraints/nonlinear/assumeconvex` to `True`. Table 2.9 summarizes the key differences in the basic algorithms, implementation of branch-and-cut routines and the performance of these solvers on the test set $TS$. Unlike earlier tables, the SGM of the wall clock times is computed over the instances solved by the particular solver and does not depend on any other solver. We see that all solvers benefit from parallelization, although without good scalability. We also see that OA with a state-of-the-art branch-and-cut MILP solver performs better than the QG algorithm with one's own branch-and-cut implementation that may lack several key MILP features. Implementing QG using callbacks to a fast commercial MILP solver seems to be the best option. This option is, however, encumbered by the availability and licensing of the MILP solver. QG with enhanced linearization schemes with one's own branch-and-cut is seen to be the next best option.

To conclude, parallel extensions of the algorithms NLP-BB and QG can accelerate their sequential versions by about 40-50% using 16 threads. The speedup is higher for difficult instances. We see some scope of improvement here as the number of nodes processed increased by only about 60% when using 16 threads. Lastly, improvements in the techniques for MILP seem to have a big impact on the methods. MINLP solvers will gain a lot if the underneath MILP solver or the branch-and-cut implementation is improved. The scope for improvement seems especially high for the academic and open-source solvers currently available.

**Table 2.9:** Comparison of algorithms deployed by different solvers along with the SGM of wall clock times and number of instances solved from set $TS$.

| | | | | one thread | | 16 threads | |
|---|---|---|---|---|---|---|---|
| solver | algo-rithm | relax-ation | branch-and-cut\bound implementation | # solved | wall time | # solved | wall time |
| *mcbnbSRel* | NLP-BB | NLP | own | 237 | 31.23 | 260 | 25.24 |
| *fscip* | QG | LP | own | 276 | 14.99 | 273 | 5.93 |
| *shot* | QG | LP | MILP solver | 309 | 8.75 | 309 | 6.40 |
| *mcqgHyb* | QG | LP | own | 295 | 17.52 | 300 | 12.66 |
| *lstoaO* | QG | LP | MILP solver | 307 | 12.32 | 325 | 6.66 |
| *oaSol* | OA | MILP | MILP solver | 295 | 11.63 | 309 | 8.19 |

# Chapter 3

# Anomalies in Parallel Branch-and-Bound Based Algorithms for MINLP

In this chapter, we study the so called 'anomalies' in parallel tree-search algorithms. Anomalies refer to the unpredictable performance of parallel algorithms with respect to the number of processors used. This includes the case when parallel algorithms run disproportionately slower or faster than their sequential counterparts. We particularly study conditions when a parallel algorithm is theoretically guaranteed to be at least as fast as the sequential algorithm. Our study concentrates on NLP-BB and QG algorithms for convex MINLP.

A formal analysis of parallel tree-search based algorithms is attributed to **?**, who introduced anomalies in branch-and-bound algorithms. Their results were improved by **?** and later by **?**, who proposed conditions to avoid anomalies. We extend these results for avoiding anomalies in NLP-BB and QG.

Let $k$ denote the number of processors used by the algorithm and $\epsilon$ be the optimality gap tolerance (the deviation from an optimal solution within which a feasible solution is acceptable as optimal). We consider the following assumptions, similar to those presented by **?** and **?** to simplify our analysis.

**Assumption 3.1.** *The processors operate 'synchronously', that is, in each iteration, at most $k$ open nodes are selected based on a heuristic function $h(.)$ and solved simultaneously.*

Note that Assumption 3.1 is equivalent to assuming that the time for solving a node and inserting a subproblem in the memory are same for all processors

at every iteration. While the solving times for nodes can vary in practice, this assumption helps establish the notion of an iteration. We refer to an iteration as one cycle of all operations executed by a processor including presolving, node processing, branching, adding cuts, checking stopping conditions, inserting new subproblems in the memory, etc. This assumption helps in analyzing the performance of parallel branch-and-bound algorithms using the number of iterations taken by the algorithm (**?**), instead of the wall clock time taken (which may vary depending on the hardware, load on the system and other factors). We denote the number of iterations by $T(k, \epsilon)$.

**Assumption 3.2.** *All idle processors are used only to solve subproblems.*

Assumption 3.2 encapsulates that processors are not allowed to remain idle, unless there are no open subproblems to be solved. Also, processors are not used to run other algorithmic tasks such as running primal heuristics, presolving, etc. when idle.

**Assumption 3.3.** *All the subsolvers used within the algorithm (for solving subproblems) are deterministic.*

Assumption 3.3 ensures that results are replicable if same initial conditions are provided to the subsolver (for example, an LP or an NLP solver) used within the algorithm. This assumption is not unrealistic, subject to the use of sufficiently small tolerance values, as observed in our numerical results.

Now, we present the definitions of anomalies and other results from **?**.

**Definition 3.0.1.** *(**?**) A behaviour exhibited by a parallel tree-search algorithm using k processors is an:*

- *acceleration anomaly, if,* $T(k, \epsilon) < \frac{T(1,\epsilon)}{k}$,
- *deceleration anomaly, if,* $\frac{T(1,\epsilon)}{k} < T(k, \epsilon) < T(1, \epsilon)$,
- *detrimental anomaly, if,* $T(k, \epsilon) > T(1, \epsilon)$.

A detrimental anomaly is depicted in Figure 3.1 and Figure 3.2. The number of iterations required are more when two threads are used (5 iterations) compared to the sequential version (3 iterations). As mentioned by **?**, one of the main reasons for such anomalies is the ambiguous selection of nodes in the sequential and the parallel versions. However, anomalies can also arise due to the ambiguity in other components of branch-and-bound algorithms. In fact, the tree might evolve

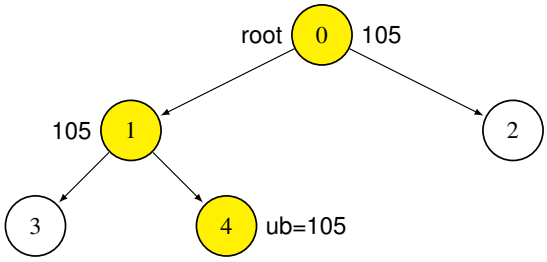



**Figure 3.1:** A sequential branch-and-bound tree ($T = 1$). The algorithm processes nodes $0, 1, 4$ respectively and then terminates.

| Iteration | Node (id) processed | | |
|---|---|---|---|
| | T=1 | T=2 | |
| | `thread0` | `thread0` | `thread1` |
| 1 | 0 | 0 | – |
| 2 | 1 | 1 | 2 |
| 3 | 4 | 9 | 10 |
| 4 | – | 11 | 12 |
| 5 | – | 13 | 14 |

**Table 3.1:** Node solved at each iteration of a sequential tree-search (shown under the column T=1), and another tree solved using two threads (column T=2).

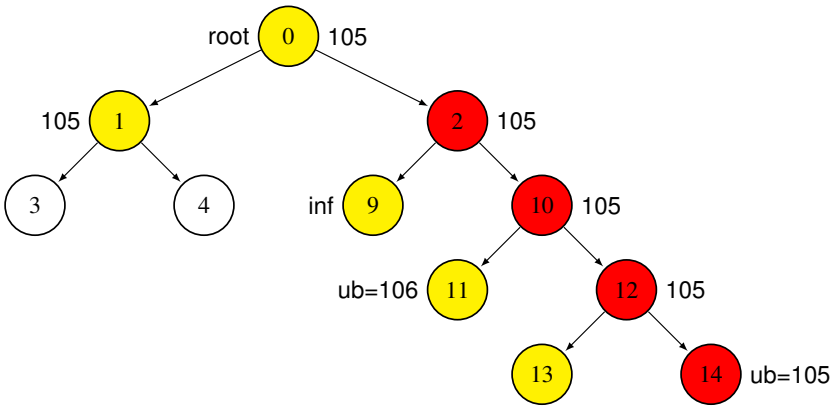



**Figure 3.2:** A branch-and-bound tree explored using two threads ($T = 2$). Two nodes are solved in parallel in each iteration except the first iteration. `thread0` solves the yellow coloured nodes and `thread1` solves the red coloured nodes. The algorithm terminates after node 14 is processed.

differently in case of ambiguous algorithmic components, hence, they must be appropriately addressed to avoid anomalies.

Sufficient conditions to avoid detrimental anomalies and necessary conditions to allow acceleration anomalies were proposed by **?**. The following definitions follow in relation to the former.

**Definition 3.0.2.** *(?) Given a set of open nodes, $\boldsymbol{P}$, a heuristic node selection function $h(.)$ is referred to as unambiguous if it satisfies the following two properties.*

1. $h(P_i) \neq h(P_j)$, *for all* $P_i, P_j \in \boldsymbol{P}$,

2. $h(P_i) \leq h(P_j)$, *for all 'descendant' nodes* $P_j$ *of* $P_i$.

All nodes that are encountered when moving down the tree along the edges starting from a node are referred to as the descendants of this node. Similarly, the

nodes encountered while moving up the tree are called the 'ancestor' nodes. A node with a lower heuristic function value has a higher priority.

**Definition 3.0.3.** *(?) A basic node is one with the minimum heuristic node selection function value at an iteration.*

If $h$ is unambiguous, then there can be only one basic node at an iteration. The following theorem states that if the node selection function is unambiguous, then detrimental anomalies are avoided.

**Theorem 3.0.4.** *(?) If $h$ is unambiguous, then $T(k,0) \leq T(1,0)$.*

The proof of Theorem 3.0.4 is based on the following facts.

- At least one node from the sequential search tree is processed in the parallel search.
- The parallel search terminates in the same or less number of iterations as the serial search.
- If $P_i$ is a basic node, then for any node $P_j$ such that $h(P_j) \leq h(P_i)$, $P_j$ must be either solved or terminated when $P_i$ is solved.

While **?** do not consider unambiguity explicitly in branching, **?** mention in their Assumption (A1) that the branching scheme applied at a node $P_i$ depends only on the information obtained along the path from $P_i$ to the root node $P_0$. However, the analysis presented by **?** focusses on best-first node selection rules, not on the branching functions. In this work, we explicitly address the unambiguity of node selection and branching functions at an iteration of the branch-and-bound algorithm. We also present a simple tie-breaking rule for both the node selection and branching variable selection.

## Parallel Branch-and-Bound Framework in Minotaur

We first discuss the parallel branch-and-bound framework, implemented within Minotaur (**?**), a generic framework for implementing MILP and MINLP algorithms. As mentioned briefly in Section 2.3, Minotaur has various algorithmic components like `NodeRelaxer`, `NodeProcessor`, `NLPEngine`, etc., as classes. The parallel branch-and-bound framework is flexible and can be customized to obtain a parallel MILP or a global solver based on branch-and-bound. We focus on solving convex MINLPs using parallel branch-and-bound based algorithms.

Although the branch-and-bound framework seems fairly simple, practically, the performance of a sequential or parallel versions of these algorithms depend on the way they have been implemented on a software platform. Different MILP and MINLP solvers use distinctive data structures, classes, subsolvers, etc., which make each implementation/solver unique in its own way. The parallel implementation of NLP-BB in Minotaur uses available classes in Minotaur and with minimum deviations from the sequential versions of the algorithm. As already described in Section 2.3.1, a single pool of open nodes is maintained by the class `TreeManager` and a prespecified number of threads solve different nodes simultaneously until the node pool is empty. Algorithm 3.1 shows the pseudocode of the parallel branch-and-bound scheme in Minotaur. Algorithm 3.2 describes the function *GetProblem*() used in Algorithm 3.1. Parallel extensions of NLP-BB and QG are already implemented in Minotaur, as explained in Section 2.3.1 and Section 2.3.3.

## 3.1 Opportunistic Parallel Branch-and-Bound in Minotaur

In this section, we describe new parallel extensions of the NLP-BB and the QG algorithms where we solve tree-nodes in parallel, though in a bit more compact way compared to the algorithms presented recently by **?**. Both these extensions are more 'opportunistic' than the ones of the same algorithms presented in Section 2.3.1 and Section 2.3.3. We use the term opportunistic in two respects: first, the threads attempt to take a new open node as soon as they solve a node, without waiting for the other threads; second, to label the algorithm as 'not deterministic' in terms of reproducibility of results.

### 3.1.1 Parallel NLP-BB

Algorithm 3.3 shows the pseudocode for the opportunistic parallel extension of NLP-BB in Minotaur. This algorithm is referred to as parallel opportunistic *mcbnb*. The minor but subtle change compared to the version presented by **?** is that this algorithm does not use the OpenMP `for` loops, thus avoiding the implicit synchronization of the threads at the end of these loops. This implies that if a thread finishes processing a node earlier than the other threads, it will not have to wait for other threads to finish processing their nodes. We note that checking the stopping conditions is easier when using `for` loops than the `while` loops

**Algorithm 3.1:** Parallel NLP-BB algorithm for convex MINLP

```
1  Set the upper bound, U = ∞ and initialize the list of open problems with
     the root NLP, H = {NLP_0}.
2  Set the state of a thread, S_t = idle, ∀ t ∈ 1 ... T.
3  Set shouldRun = true.
4  while H ≠ ∅ AND shouldRun == true do
5      for t from 1 to T do
6          if S_t = idle then
7              GetProblem().
8          if S_t == assigned then
9              Solve NLP_t at thread t.
10             if NLP_t is optimal and the optimal value f(x̂^t) > U then
11                 NLP_t can be pruned on the basis of bound.
12             else if x̂^t ∈ ℤ, ∀ i ∈ I then
13                 Update U ← min{U, f(x̂^t)}.
14                 GetProblem().
15             else if NLP_t is infeasible then
16                 Prune the node.
17                 GetProblem().
18             else
19                 Use a branching rule to generate NLP_{t_1} and NLP_{t_2}.
20                 Update the list: H = H ∪ {NLP_{t_1}, NLP_{t_2}}.
21                 GetProblem().
22     if stopping conditions met then
23         Set shouldRun = false.
```

due to the implicit synchronization at the end of one solving cycle or iteration. In the current algorithm, we adjust the stopping conditions by also counting the number of nodes assigned to the threads (that have been removed from the node pool) in addition to the size of node pool.

### 3.1.2 Parallel QG

The parallel schematic for the opportunistic parallel extension of QG is similar to that of NLP-BB. Algorithm 3.4 presents a pseudocode for this algorithm.

**Algorithm 3.2:** Getting a problem from the list of problems for a given $t \in \{1, \ldots, T\}$.

```
if H ≠ ∅ then
    Remove an NLP from H as per the search strategy.
    Assign NLP to thread t, NLP_t ← NLP and set S_t = assigned.
else
    Set the status of thread, S_t = idle.
```

**Algorithm 3.3:** Opportunistic parallel NLP-BB algorithm in Minotaur.

```
Initialize upper bound, U = ∞, state of thread, S_t = idle, parameter to
  indicate if a node is assigned to a thread t, K_t = 0, t = 1, 2 ..., T.
Add the root NLP relaxation to the pool of open nodes, H.
while H ≠ ∅ or ∑_{t=1}^{T} K_t > 0 do
    K_t ← 0.
    if S_t = idle and H ≠ ∅ then
        GetNode().
        K_t ← 1.
    if S_t = assigned then
        Solve NLP_t at thread t.
        if NLP_t is optimal and (x̂^t)_i ∈ ℤ, ∀ i ∈ I then
            Update U ← min{U, f(x̌^t)}.
        else if NLP_t is infeasible then
            Prune this node.
            GetNode().
            K_t ← 1.
        else
            Branch: generate two NLP subproblems and add to H.
            GetNode().
            K_t ← 1.
```

It has two main differences from the opportunistic *mcbnb*. First, LPs (instead of NLPs) are solved as nodes and second, additional linearization cuts are added at nodes that yield integer solutions. The function *GetProblem*() for this algorithm

is similar to that shown in Algorithm 3.2, except that the node pool contains LPs instead of NLPs.

**Algorithm 3.4:** Opportunistic parallel QG (LP/NLP based branch-and-bound) algorithm in Minotaur.

1 Initialize upper bound, $U = \infty$, state of thread, $S_t = idle$, cut pool for each thread, $C_t = \emptyset,\ \forall\ t \in 1 \ldots T$, parameter to indicate if a node is assigned to a thread $t$, $K_t = 0$.
2 Add root LP relaxation to the pool of open nodes $\mathcal{H}$.
3 **while** $\mathcal{H} \neq \emptyset$ *or* $\sum_{t=1}^{\top} K_t > 0$ **do**
4  $K_t \leftarrow 0$.
5  **if** $S_t = idle$ *and* $\mathcal{H} \neq \emptyset$ **then**
6  *GetNode()*.
7  $K_t \leftarrow 1$.
8  **if** $S_t = assigned$ **then**
9  Add *new* cuts from $C_t,\ \forall t$ in $1 \ldots T$, to $LP_t$.
10  Solve $LP_t$ at thread $t$.
11  **if** $LP_t$ *is optimal and* $(\hat{x}^t)_i \in \mathbb{Z},\ \forall\ i \in \mathcal{I}$ **then**
12  Solve F-NLP($\hat{x}^t$), let the point returned be $\check{x}^t$.
13  **if** *F-NLP*($\hat{x}^t$) *is optimal* **then**
14  Update $U \leftarrow \min\{U, f(\check{x}^t)\}$.
15  Generate linearizations of all nonlinear constraints violated by $\hat{x}^t$, at $\check{x}^t$, and add to $C_t$ and $LP_t$.
16  Go to step 10.
17  **else if** $LP_t$ *is infeasible* **then**
18  Prune this node.
19  *GetNode()*.
20  $K_t \leftarrow 1$.
21  **else**
22  Branch: generate two *LP* subproblems and add to $\mathcal{H}$.
23  *GetNode()*.
24  $K_t \leftarrow 1$.

## 3.2 Parallel NLP-BB with No Detrimental Anomalies

We extend Theorem 3.0.4 in the context of MINLP algorithms in this section. State-of-the-art MINLP algorithms like NLP-BB comprise various important components in addition to node-selection heuristic functions that have to be unambiguous for the MINLP algorithm to be nondetrimental. Branching is one of the most important components that need to be addressed. We focus on unambiguous branching functions to derive a nondetrimental parallel NLP-BB algorithm.

### 3.2.1 Unambiguous Branching Functions

Since a node (subproblem) is constructed by explicitly adding a set of branching constraints to its parent, the notion of unambiguity must also include branching in addition to unambiguous node selection heuristic functions. We first extend the notion of unambiguity defined in Definition 3.0.2 by formally defining unambiguous branching functions, based on widely used simple variable disjunctions. These disjunctions are expected to umambiguously select a branching candidate from a subset of $\mathcal{I}$ (the set of all integer constrained variables). This definition can be easily extended to cover more general branching functions.

**Definition 3.2.1.** *Consider an open node $P_i \in \boldsymbol{P}$ that has not been pruned after processing. Let $x^*$ denote the optimal solution of $P_i$ and the set of branching candidates be $\mathcal{I}_C := \{j \in \mathcal{I} : x^*_j \notin \mathbb{Z}\}$. A branching variable selection function $\nu(.)$ over $\mathcal{I}_C$ at $P_i$ is referred to as unambiguous, if*

- *$\nu$ is deterministic and depends only on the information obtained along the path from $P_i$ to the root node $P_0$,*
- *$\nu(j) \neq \nu(k)$ for all $j, k \in \mathcal{I}_C$, $j \neq k$.*

It can be easily verified that the simple lexicographic branching rule satisfies Definition 3.2.1. Strong branching involves partly or fully solving an NLP subproblem, and if a deterministic subproblem solving algorithm is used, strong branching also satisfies Definition 3.2.1. An unambiguous branching function is essential for unambiguous node creation in the tree.

Next, we present an unambiguous branching scheme, a variant of the reliability branching scheme, that is presumably more effective than the lexicographic branching and less expensive than full strong branching. We also discuss some implementation related issues.

### 3.2.2 Unambiguous Reliability Branching Scheme

We propose a simple reliability branching based scheme, termed as *ancestRel* branching, to unambiguously select a branching candidate at a node.

**Reliability Branching** First, we briefly explain reliability branching, a hybrid scheme that balances strong branching and pseudocost branching; see **?** for a detailed description of each of these. Let $x^*$ be the optimal solution of the node in consideration. Strong branching scheme estimates the potential dual bound improvement if some $x_j \in \mathcal{I}_C$ is selected as the branching candidate, by fully or partially solving two subproblems; one with the down-branch ($x_j \leq \lfloor x_j^* \rfloor$) and the other with the up-branch ($x_j \geq \lceil x_j^* \rceil$) applied to this node, and selects the most promising candidate. Typically, strong branching at each node is deemed expensive because for each branching candidate, one has to (fully or partly) solve two subproblems for each branching candidate.

Pseudocost branching is another scheme that evaluates different candidates based on a score corresponding to each integer variable. This score is updated based on the dual bound improvement corresponding to each $x_j \in \mathcal{I}_C$, per unit change in the value of $x_j$. Let $\delta^{up} := \lceil x_j^* \rceil - x_j^*$, $\Delta_j^{up} := f_j^{up} - f^*$ and $\zeta_j^{up} := \frac{\Delta_j^{up}}{\delta_j^{up}}$ represent this change along the up-branch where $f^*$ is the optimal objective function value at the node. Let $\sigma^{up}$ denote the sum of $\zeta_j^{up}$ over a number of nodes, $n_j^{up}$, where $j$ is chosen as the branching variable and the subproblem with the up-branch is feasible. Then, the up-pseudocost at a node is given by $\psi^{up} := \frac{\sigma^{up}}{n_j^{up}}$. The down-pseudocost is calculated in a similar way. The best candidate is decided based on a score,

$$s_j := \alpha \min\{\psi_j^{up}\delta_j^{up}, \psi_j^{down}\delta_j^{down}\} + (1-\alpha)\max\{\psi_j^{up}\delta_j^{up}, \psi_j^{down}\delta_j^{down}\}, \tag{3.1}$$

where $\alpha \in (0,1)$ is a parameter, typically set to $\frac{1}{6}$. The drawback with this scheme is that the scores do not provide good estimates of the possible dual bound improvement near the root of the tree, when less nodes have been solved.

The reliability branching scheme is a hybrid combination of the strong branching and the pseudocost branching schemes, and attempts to overcome the drawbacks of both of them. It does limited (and dynamic) strong branching in the tree, and more so near the root node to obtain a reasonable initialization of pseudocosts, and then keeps updating the scores along the tree using the node-solve information.

***ancestRel* Branching** This branching uses pseudocosts from only the ancestors of the node in consideration for calculating scores of different branching candidates. It is not difficult to comprehend that such a scheme would preserve the unambiguity property, because the same child nodes would be generated by branching at a node, irrespective of the number of processors used by the algorithm. Also, this scheme uses a part of node solving information generated in the tree, which might yield a better branching candidate compared to other simple unambiguous branching schemes like lexicographic branching. Figure 3.3 illustrates *ancestRel* branching scheme in a pictorial form. Detailed steps of the *ancestRel* branching function are shown in Algorithm 3.5. $D_{max}$ represents a depth level after which pseudocosts are used and strong branching is avoided. $d_{min}$ is the threshold (node) distance used to compare against the number of iterations passed since last strong branching on a candidate. $l_j^{str}$ denotes the latest iteration number at which a variable $x_j$ was strong branched and $N$ represents the total number of nodes solved till the current iteration. $n_{rel}$ represents a reliability threshold: if a variable is branched (up and down) more than $n_{rel}$ number of times, it is considered a reliable candidate, that is, its pseudocosts are trusted in lieu of strong branching on it.

A similar branching scheme, the *privateRel* brancher has been presented by **?**, but it uses pseudocosts from all the nodes processed by a thread in the tree and could result in different scores at a node when using different number of threads. Hence, one can not claim that the same node is created when different number of threads are used. On the other hand, the scores computed by *ancestRel* brancher at a node remain unaffected by the number of threads used in the algorithm.

**Proposition 3.2.2.** *The ancestRel branching variable selection strategy with the lexicographic tie-breaking rule is unambiguous.*

*Proof.* If a node does not get created when using a particular number of processors, then there is no branching required. Otherwise, since the pseudocosts from only the ancestor nodes are used to calculate scores, the selected branching candidate at a node is independent of other nodes in the tree, and the number of processors used to generate this tree. Also, the set of branching candidates, $\mathcal{I}_C$, is unambiguous because the (NLP) subsolver used is deterministic, and a lexicographic tie-breaking rule is sufficient to select a unique branching candidate each time such a node is created and branched upon. Hence, the conditions of Definition 3.2.1 are satisfied by this branching scheme. □

**Algorithm 3.5:** Selecting a branching candidate at a node using the *ancestRel* brancher

**Input :** A set of branching candidates $\mathcal{I}_C \subseteq I$, parameters $d_{min}, N, n_{rel}, d_{max}, n_{str}, w_s, w_i$.

1 **for** $j \in \mathcal{I}_C$ **do**
2  **if** $d_{min} > |N| - l_j^{str}$ *OR* $n_j^{up} >= n_{rel}$ *AND* $n_j^{down} >= n_{rel}$ **then**
3  Add $j$ to $\mathcal{I}_C^{rel}$ (the set of reliable candidates in $\mathcal{I}_C$).
4  **else**
5  Set $s_j = n_j^{up} + n_j^{down} - w_s(\psi_j^{up} + \psi_j^{down}) - w_i \max\{\Delta_j^{up}, \Delta_j^{down}\}$ Add $j$ to $\mathcal{I}_C^{unrel}$ (set of unreliable candidates in $\mathcal{I}_C$).
6 Find $x_{rel} = \arg\max_{x_r \in \mathcal{I}_C^{rel}} s_r$, where $s_r = \alpha \min\{\psi_r^{up}\delta_r^{up}, \psi_r^{down}\delta_r^{down}\} + (1-\alpha)\max\{\psi_r^{up}\delta_r^{up}, \psi_r^{down}\delta_r^{down}\}$.
7 Find $n_{unrel} := n_{str}$ if $D_{node} \leq D_{max}$, otherwise 0.
8 Pick a maximum of $n_{unrel}$ candidates from $\mathcal{I}_C^{unrel}$ in the order of decreasing $s_j$ values and form the set $\bar{\mathcal{I}}_C^{unrel}$.
9 **for** $u \in \bar{\mathcal{I}}_C^{unrel}$ **do**
10  Strong branch on $x_u$, and update $\psi_u^{up} = \frac{1}{n_u^{up}+1}(\psi_u^{up} * n_u^{up} + \frac{\Delta_u^{up}}{\delta_u^{up}})$ and $\psi_u^{down} = \frac{1}{n_u^{down}+1}(\psi_u^{down} * n_u^{down} + \frac{\Delta_u^{down}}{\delta_u^{down}})$.
11  Update $l_u^{str}, n_u^{up}, n_u^{down}$.
12 Find $x_{unrel} = \arg\max_{x_u \in \mathcal{I}_C^{unrel}} s_u$, and return $x^{best} = \arg\max_{x_{unrel}, x_{rel}} (s_{x_{rel}}, s_{x_{unrel}})$.

From the implementation perspective, *ancestRel* branching incurs a small storage and operations overhead because one needs to store the pseudocosts and other related information at a node along with the description of that node in the tree. The following two ways can be used to implement this scheme.

1. One can store branching candidates and the respective pseudocosts, times branched and 'last_strong_branched' information along with each node. Whenever branching is required at node, consolidate and pass all the information to the children and delete the same from this node. In this way, all open nodes will have aggregated scores for a subset of integer variables stored with them, but there would be additional delete operations after each branching.

2. Instead of consolidating all the ancestral pseudocost information, store only the local information at each node. While branching at a node, query all ancestor nodes up to the root node and then compute the aggregated scores.

In this implementation, all nodes will have local information stored with them.

We have chosen the former option in our implementation. At present, we do not delete any information from the parent nodes and have not encountered any memory related issues in our computational experiments, but we plan to accomplish this feature, as well as the node local information based branching in future releases of Minotaur.

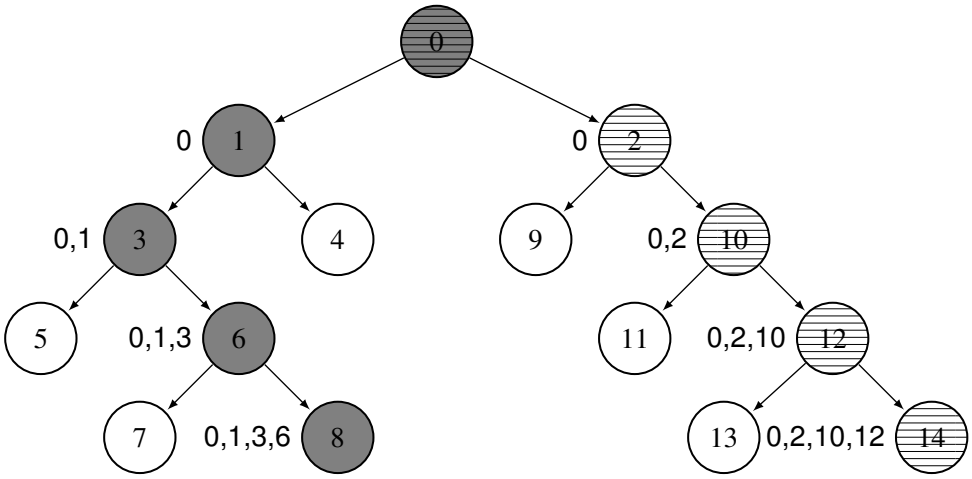



**Figure 3.3:** Illustration of unambiguous reliability branching when using two threads. The root node indexed 0 and then the gray-colored nodes are solved by `thread0` and the hatched nodes (except 0) are solved by `thread1`. Both the threads use pseudocosts from their ancestor nodes: `thread0` uses pseudocosts only from the gray-colored nodes and `thread1` from only the hatched nodes (indices are shown on the left of each node).

### 3.2.3 Unambiguous Node Selection

While searches like depth-first, width-first or best-first have been discussed in the past (**??**), we show that a practically effective node selection strategy called the best-then-dive strategy, coupled with a simple tie-breaking rule is unambiguous and can avoid detrimental anomalies. The best-then-dive strategy first selects a node with the best lower bound, and then keeps diving (processing one of the two immediate child nodes) until a node is pruned. At this stage, it looks for a node with the best known lower bound to be processed next. If there are multiple nodes with the same bound value, then a tie-breaking score can be used as a secondary key for prioritizing the candidate nodes. Let $P_i$ be a node that has just been processed and branched, and has an optimal value $\hat{z}$. Assuming that

the diving preference is towards the left node, we define the primary heuristic function value for the child nodes of $P_i$ as follows.

$$h(P_j) = \begin{cases} -\infty, & \text{if } P_j \text{ is the left child of } P_i, \\ \hat{z}, & \text{otherwise.} \end{cases} \qquad (bTd)$$

In case a node gets pruned, an open node with the best lower bound value $\hat{z}$ is selected. Very often, multiple such nodes exist, for which an unambiguous tie-breaking mechanism is required. Consider a node $P_i$ with a score $s$ and that two child nodes have to be created using a branching variable disjunction. A score $2s$ is assigned to the left child of $P_i$ (generated using the $\leq$ disjunction) and $2s + 1$ to the right. We will refer to this scoring mechanism as the $2s\_2s + 1$ rule. The root node is assigned a score 1.

**Proposition 3.2.3.** *The best-then-dive node selection strategy with the* $2s\_2s + 1$ *tie-breaking rule is unambiguous.*

*Proof.* We require to show that both the conditions of Definition 3.0.2 are always satisfied by the mentioned scheme. At any iteration, if a processed node is branched, the left child node clearly has the unique lowest primary function value, so does its child until a node is pruned which complies with Definition 3.0.2. Again, suppose that a node is pruned and another node has to be selected. Either there exits a single node with a lowest lower bound value, or by construction, the tie-breaking rule serves as the secondary key, which is distinct for any two nodes in the tree. In both the cases, conditions of Definition 3.0.2 are satisfied. □

### 3.2.4 Nondetrimental NLP-BB

We denote by *mcbnbDeter*, the parallel extension of NLP-BB in Minotaur that uses the following unambiguous components.

- best-then-dive node selection strategy with the $2s\_2s + 1$ tie-breaking rule
- *ancestRel* branching strategy with the lexicographic tie-breaking rule
- a deterministic NLP solver.

In addition, we require synchronization of threads at various points in the algorithm, for example to pass (unambiguous) initial conditions to the subsolvers,

as well as unambiguity on other algorithmic components. For example, the presolving of nodes in Minotaur (by default) is done based on a user defined presolve frequency, where the 'id' of a node is used to check if this particular node must be presolved or not. But the id of a node in the tree might vary when number of processors deployed is varied. Hence, we adjusted it to be dependent on the $2s\_2s + 1$ tie-breaking score of the nodes (which is a tree- invariant identifier for a node). Additionally, we disabled features such as 'guided diving', which determines when to create branches based on the best solution so far and could cause ambiguity when creating branches during an iteration.

**Theorem 3.2.4.** *The algorithm mcbnbDeter satisfies $T(k) \leq T(1)$ for $k > 1$.*

*Proof.* The result follows by extending Theorem 3.0.4 to include the unambiguity of all the involved components: the node selection function (Proposition 3.2.3), the branching function (Proposition 3.2.2) and the NLP solver (Assumption 3.3). □

## 3.3 Parallel QG with No Detrimental Anomalies

In this section, we formally define the notion of unambiguity for an integral component of branch-and-cut based algorithms, the cutting planes.

**Definition 3.3.1.** *A vector valued cut generating function $\pi(.)$ at a node $P_i$ is referred to as unambiguous if $\pi$ is deterministic and depends only on the information obtained along the path from $P_i$ to the root node $P_0$.*

We consider the case when a cut generating function uses only a point $\hat{x} \in \mathbb{R}^n$ obtained at an LP node, $P_i$, by solving a fixed NLP, as is done in QG. The point $\hat{x}$ is either an integer feasible solution to (P) or a point from a feasibility problem that minimizes some measure of constraint violation at this node. We refer to the strategy that uses this cut generating function as *cutGenQG*.

**Proposition 3.3.2.** *The cut generation strategy cutGenQG is unambiguous.*

*Proof.* Since *cutGenQG* uses only a point returned from a deterministic LP or an NLP solver at a node, this information depends only on the node in consideration. Hence, conditions of Definition 3.3.1 are satisfied by *cutGenQG*. □

Next, we denote by *mcqgDeter*, the parallel extension of QG in Minotaur that uses the following unambiguous components.

- best-then-dive node selection strategy with the $2s\_2s+1$ tie-breaking rule
- *ancestRel* branching strategy with the lexicographic tie-breaking rule
- *cutGenQG* cutting plane strategy
- a deterministic NLP solver
- a deterministic LP solver

**Theorem 3.3.3.** *The algorithm mcqgDeter satisfies* $T(k) \leq T(1)$ *for* $k > 1$.

*Proof.* The result follows by extending Theorem 3.0.4 to include the unambiguity of all the involved functions in algorithmic components: the node selection function (Proposition 3.2.3), the branching function (Proposition 3.2.2), the cut generating function (Proposition 3.3.2), and the LP and NLP solvers (Assumption 3.3). □

## 3.4 Reproducibility in Parallel NLP-BB and Parallel QG

The use of unambiguous algorithmic components in parallel variants of NLP-BB and QG automatically results in deterministic behaviour of these algorithms. Reproducibility in parallel algorithms is a desired feature due to various reasons including performance analysis, debugging during code development, etc. In Minotaur, we implement a function that synchronizes the operations of various classes used by the branch-and-bound algorithms, and uses appropriate solver options that ensure unambiguity of all algorithmic components. It might be noted that unambiguity is sufficient for ensuring repeatability of results but not necessary.

## 3.5 Computational Results

We have used the same computational setup mentioned in Section 2.2 including the hardware, software, compilers, etc. Since our experiments intend to highlight the effect of parallelism on algorithms, hyperthreading is disabled. The code for all our algorithms is available in the development version of Minotaur[1]. We have used 374 instances from MINLPLib (**?**) that are known to be convex. A hard limit

[1] http://github.com/minotaur-solver/minotaur

of one hour on the wall clock time has been used for all our experiments, and all the solution times are reported in seconds.

First, we show the performance of the opportunistic parallel extension of *mcbnb* compared to its single-threaded version in Table 3.2. Each row of the top table in Table 3.2 corresponds to a different number of threads, $T = 2, 4, 8, 16$. The column '# solved by' lists the number of instances solved to optimality within the time limit by the proposed method and by both the reference solver, single-threaded *mcbnb*, (denoted *mcbnb1*) as well as the proposed method (under the column 'both'). The first column with the headings 'time' and 'nodes' shows the shifted geometric mean (SGM) of these measures reported by the reference solver for the number of instances shown in the column 'both'. The second column shows the relative SGM (denoted 'rel.') of the multithreaded variants. We have used a shift of 10 for SGM of time and 100 for the number of nodes processed. The table at the bottom shows the performance of algorithms over instances of varying difficulty.

As indicated in Table 3.2, *mcbnb16* could solve 29 additional instances compared to *mcbnb1*. Also, the time taken is reduced by more than 60%. This performance is better compared to the opportunistic schemes reported earlier by **?**. Also, performance gets better when instances of higher difficulty level are considered.

Similarly, we show the scalability graphs (explained in Section 2.3.1) in Figure 3.4 that demonstrate the performance of *mcbnb*. The plot for *mcbnb1* is a step function for which the peak value (about 0.71 in this case) indicates the fraction of instances that were solved using *mcbnb1*. The ordinate corresponding to a value at, say $2^{-1}$, indicates the fraction of instances that could be solved by a multi-threaded variant by a factor of two or more as compared to *mcbnb1*. For instance, *mcbnb16* solves 40% of the instances at least two times faster than *mcbnb1*. Similarly, *mcbnb4* and Also, the rightmost values on the plots show the fraction of instances that could be solved within the time limit.

For the instances shown in Table 3.3, at least one multithreaded variant takes more time compared to *mcbnb1* demonstrating a detrimental anomaly in terms of wall clock time. However, the instances of the `jbearing` class require solving only one node, hence the time taken by the multithreaded variants is mostly the initial setup time. Table 3.3

For the deterministic *mcbnb*, the performance is shown in Table 3.4. These results are obtained using the following options in Minotaur: *–brancher ancestRel*

**Table 3.2:** (Top) Comparison of opportunistic *mcbnb1* with opportunistic *mcbnb* using multiple threads. *mcbnb1* could solve 268 instances. (Bottom) Break-up of performance of *mcbnb16* over instances of varying difficulty.

| # threads | # solved by | | wall time | | nodes | |
|---|---|---|---|---|---|---|
| (T) | *mcbnbT* | both | *mcbnb1* | rel. | *mcbnb1* | rel. |
| 2 | 273 | 268 | 27.93 | 0.77 | 247.43 | 1.16 |
| 4 | 275 | 268 | 27.93 | 0.56 | 247.43 | 1.19 |
| 8 | 287 | 268 | 27.93 | 0.45 | 247.43 | 1.26 |
| 16 | 297 | 268 | 27.93 | 0.36 | 247.43 | 1.34 |

| | # solved | wall time | | nodes | |
|---|---|---|---|---|---|
| time | by both | *mcbnb1* | rel. | *mcbnb1* | rel. |
| > 0 | 268 | 27.93 | 0.36 | 247.43 | 1.34 |
| > 10 | 131 | 117.71 | 0.23 | 883.07 | 1.46 |
| > 100 | 66 | 397.13 | 0.17 | 2001.69 | 1.61 |
| > 500 | 27 | 981.54 | 0.14 | 5160.02 | 1.83 |

**Table 3.3:** Instances for which the wall clock time taken by *mcbnbOppor* does not always improve when using more processors.

| name | *mcbnbOppor1* | *mcbnbOppor2* | *mcbnbOppor4* | *mcbnbOppor8* | *mcbnbOppor16* |
|---|---|---|---|---|---|
| jbearing100 | 4.68 | 6.34 | 9.62 | 15.92 | 25.11 |
| jbearing25 | 0.71 | 0.94 | 1.51 | 2.6 | 3.96 |
| jbearing50 | 1.8 | 2.39 | 3.7 | 6.19 | 9.49 |
| jbearing75 | 3.18 | 4.22 | 6.51 | 11.49 | 16.67 |
| slay08m | 16.12 | 47.35 | 36.89 | 22.7 | 18.62 |
| slay09h | 69.7 | 285.41 | 346.38 | 235.7 | 259.8 |
| slay09m | 29.37 | 109.38 | 111.55 | 48.64 | 54.11 |
| slay10h | 572.52 | 1817.39 | 2046.0 | 2008.42 | 557.54 |
| slay10m | 215.87 | 581.38 | 939.65 | 762.47 | 743.05 |

*–tb_rule* $2s\_2s+1$ *–mcbnb_deter_mode 1*. 'Guided diving' was not disabled in these runs. A couple of columns at the end in Table 3.4 under the title "iters" show the number of iterations taken by each multithreaded variant. The scalability graphs in terms of wall clock time are shown in Figure 3.5. As evident from Table 3.4 and Figure 3.5, the performance of *mcbnbOppor* is better than *mcbnbDeter* in terms of wall clock time because the former exploits parallelism in an opportunistic way. However, multithreaded *mcbnbDeter* variants can provide a guaranteed to not be worse than *mcbnbDeter1* and also be reproducible.

**Table 3.4:** (Top) Comparison of deterministic *mcbnb1* with deterministic *mcbnb* using multiple threads. *mcbnb1* could solve 245 instances. (Bottom) Break-up of performance of *mcbnb16* over instances of varying difficulty.

| # threads | # solved by | | wall time | | nodes | | iters | |
|---|---|---|---|---|---|---|---|---|
| (T) | *mcbnbT* | both | *mcbnb1* | rel. | *mcbnb1* | rel. | *mcbnb1* | rel. |
| 2 | 252 | 245 | 27.02 | 0.82 | 224.68 | 1.02 | 224.79 | 0.64 |
| 4 | 260 | 245 | 27.02 | 0.72 | 224.68 | 1.06 | 224.79 | 0.43 |
| 8 | 260 | 245 | 27.02 | 0.72 | 224.68 | 1.15 | 224.79 | 0.30 |
| 16 | 267 | 245 | 27.02 | 0.49 | 224.68 | 1.33 | 224.79 | 0.22 |

| time | # solved by both | wall time *mcbnb1* | rel. | nodes *mcbnb1* | rel. | iters *mcbnb1* | rel. |
|---|---|---|---|---|---|---|---|
| > 0 | 245 | 27.02 | 0.49 | 224.68 | 1.33 | 224.79 | 0.64 |
| > 10 | 117 | 120.17 | 0.33 | 791.42 | 1.33 | 791.55 | 0.15 |
| > 100 | 55 | 544.35 | 0.22 | 2039.41 | 1.24 | 2039.56 | 0.12 |
| > 500 | 30 | 1327.29 | 0.16 | 4194.64 | 1.17 | 4195.10 | 0.10 |

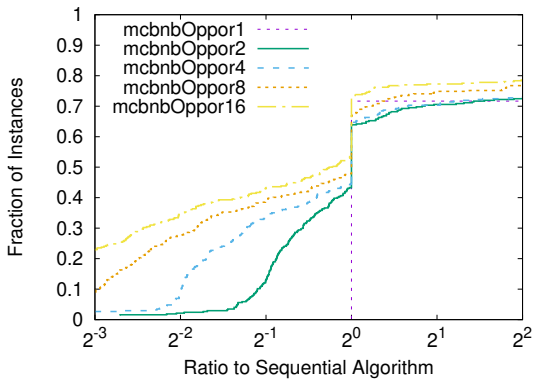



**Figure 3.4:** Scalability graphs of wall clock times taken by opportunistic *mcbnb* using different number of threads on 374 convex MINLPLib instances.

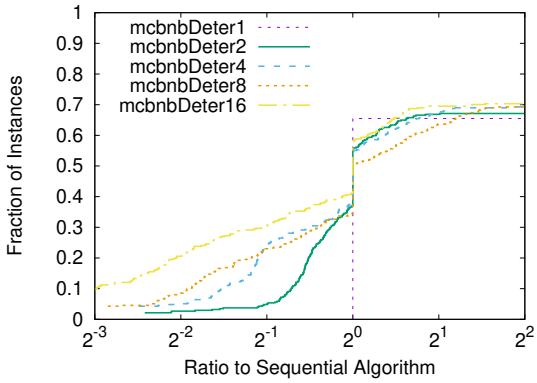



**Figure 3.5:** Scalability graphs of wall clock times taken by deterministic *mcbnb* (with guided diving enabled) on 374 convex MINLPLib instances.

We are able to completely eliminate detrimental anomalies in *mcbnb* using the above mentioned options in Minotaur in terms of the number of iterations taken by the algorithms, except the instances shown in Table 3.5. Disabling guiding diving (using `-guided_dive 0`) in Minotaur eliminates detrimental anomalies in the instances `enpro48pb`, `ex4` and `slay07m` also. The remaining few instances exhibit anomalies presumably due to various tolerances used in Minotaur and the subsolvers.

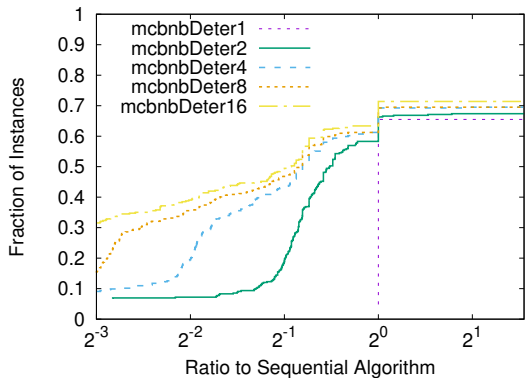



**Figure 3.6:** Scalability graphs of number of iterations taken by deterministic *mcbnb* using different number of threads on 374 convex MINLPLib instances.

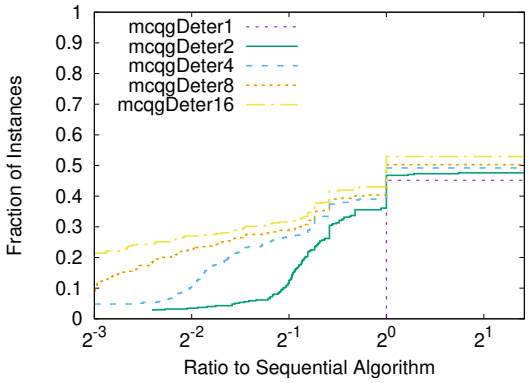



**Figure 3.7:** Scalability graphs of number of iterations taken by deterministic *mcqg* using different number of threads on 374 convex MINLPLib instances.

**Table 3.5:** *mcbnbDeter* showing anomalous behaviour in terms of the number of iterations when guided diving is not disabled.

| name | *mcbnbDeter1* | *mcbnbDeter2* | *mcbnbDeter4* | *mcbnbDeter8* | *mcbnbDeter16* |
|---|---|---|---|---|---|
| `enpro48pb` | 261 | 292 | 113 | 71 | 49 |
| `ex4` | 43 | 62 | 39 | 26 | 22 |
| `portfol_roundlot` | 1093 | 154 | 2108 | 75 | 900 |
| `rsyn0805m04h` | 94 | 97 | 84 | 39 | 30 |
| `rsyn0840m02h` | 159 | 273 | 118 | 69 | 32 |
| `slay07m` | 210 | 136 | 100 | 219 | 141 |
| `syn40m` | 16606 | 71335 | 48749 | 20827 | 9477 |

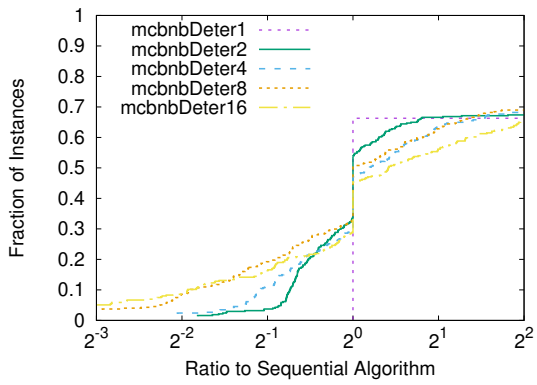



**Figure 3.8:** Scalability graphs of wall clock times taken by deterministic *mcbnb* with guided diving disabled on 374 convex MINLPLib instances.

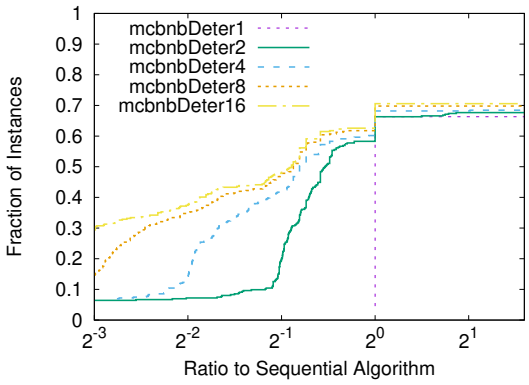



**Figure 3.9:** Scalability graphs of number of iterations taken by deterministic *mcbnb* with guided diving disabled on 374 convex MINLPLib instances.

In terms of wall clock time taken, Table 3.8 lists the instances which do not always benefit when the number of processors is increased.

**Table 3.6:** (Top) Comparison of deterministic *mcbnb1* with deterministic *mcbnb* using multiple threads when guided diving is disabled. *mcbnb1* could solve 248 instances. (Bottom) Break-up of performance of *mcbnb16* over instances of varying difficulty.

| # threads | # solved by | | wall time | | nodes | | iters | |
|---|---|---|---|---|---|---|---|---|
| (T) | *mcbnbT* | both | *mcbnb1* | rel. | *mcbnb1* | rel. | *mcbnb1* | rel. |
| 2 | 253 | 248 | 25.32 | 0.86 | 221.56 | 1.02 | 221.56 | 0.65 |
| 4 | 256 | 248 | 25.32 | 0.87 | 221.56 | 1.12 | 221.56 | 0.45 |
| 8 | 261 | 248 | 25.32 | 0.76 | 221.56 | 1.19 | 221.56 | 0.31 |
| 16 | 264 | 248 | 25.32 | 0.88 | 221.56 | 1.34 | 221.56 | 0.22 |

| | # solved | wall time | | nodes | | iters | |
|---|---|---|---|---|---|---|---|
| time | by both | *mcbnb1* | rel. | *mcbnb1* | rel. | *mcbnb1* | rel. |
| > 0 | 248 | 25.32 | 0.88 | 221.56 | 1.34 | 221.56 | 0.22 |
| > 10 | 132 | 87.66 | 0.78 | 647.97 | 1.37 | 647.97 | 0.15 |
| > 100 | 57 | 484.00 | 0.55 | 1751.29 | 1.29 | 1751.29 | 0.13 |
| > 500 | 34 | 1042.37 | 0.51 | 3196.84 | 1.35 | 3196.84 | 0.12 |

**Table 3.7:** Instances for which *mcbnbDeter* shows anomalous behaviour in number of iterations when guided diving is disabled.

| name | *mcbnbDeter1* | *mcbnbDeter2* | *mcbnbDeter4* | *mcbnbDeter8* | *mcbnbDeter16* |
|---|---|---|---|---|---|
| `netmod_kar2` | 3159 | 310 | 6382 | 232 | 83 |
| `portfol_roundlot` | 1093 | 1541 | 296 | 100 | 68 |
| `rsyn0805m02h` | 220 | 370 | 167 | 39 | 32 |
| `rsyn0805m04h` | 66 | 109 | 63 | 32 | 35 |
| `rsyn0840m02h` | 131 | 239 | 87 | 57 | 31 |
| `slay07h` | 192 | 331 | 99 | 95 | 109 |
| `syn40m` | 17628 | 71505 | 49203 | 21155 | 9746 |

Now, we present the results for opportunistic and deterministic variants of *mcqg*. Figure 3.10 shows the scalability graphs, which illustrate that *mcqgOppor16* could solve about 40% of the instances in half the time compared to *mcqgOppor1*. Table 3.9 shows the SGM of wall time taken and the number of nodes. For the most difficult instances (time > 500), improvement up to 88% was obtained, and overall, 16 additional instances could be solved. Again, these results are better compared to the opportunistic version presented in **?**.

Table 3.10 shows the instances where *mcqgOppor* exhibits anomalous behaviour in terms of wall clock time. We note that this list is longer compared to that of *mcbnb*.

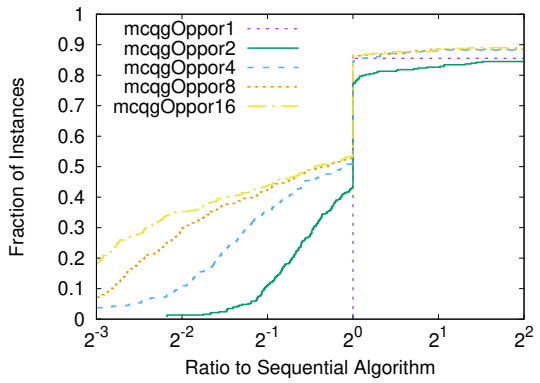



**Figure 3.10:** Scalability graphs of wall clock times taken by opportunistic *mcqg* on 374 convex MINLPLib instances.

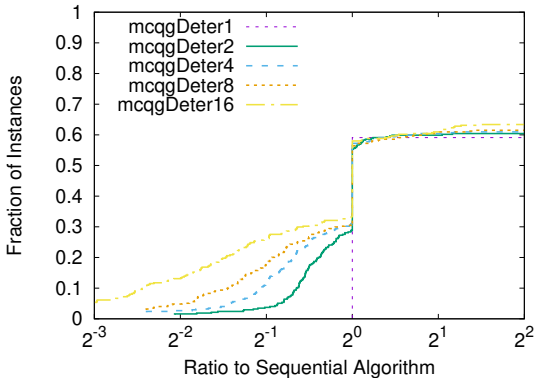



**Figure 3.11:** Scalability graphs of wall clock times taken by deterministic *mcqg* on 374 convex MINLPLib instances.

## 3.6 Conclusion and Future Research

It is important to study anomalies in parallel branch-and-bound algorithms to design better strategies in practice that can enhance scalability of parallel algorithms. We addressed detrimental anomalies in two convex MINLP algorithms, NLP-BB and QG. We extended the notion of unambiguity to variable branching functions and cut generating functions. We also showed that these theoretical ideas can be extended to practically effective algorithmic components like hybrid node selection strategies like best-then-dive (instead of pure strategies like depth-first, best-first, etc.), branchers like *ancestRel* (instead of lexicographic brancher), etc. Our computational experiments show that detrimental anomalies can be eliminated to a great extent in practical algorithms. Opportunistic versions perform better in terms of wall clock times than the deterministic versions on average, because deterministic versions tend to synchronize more and incur some extra intervals of time during the tree search. However, deterministic versions can avoid detrimental anomalies with guarantees, and can also provide reproducible results.

The analysis presented so far depends on the number of iterations. In terms of wall clock times, the opportunistic algorithms seem to perform better, hence it remains to be explored how unambiguity and speed can be achieved simultaneously. Also, unambiguity can be extended to MILP based algorithms like OA.

Another immediate extension of our work is based on Section IV-C in **?**, where coping with general parallel-to-parallel anomalies is addressed. The main reason of occurrence of these anomalies is the existence of 'imperfect' iterations, those iterations in which sufficient number of open nodes are not available for

the processors. The sufficient conditions provided by **?** depend on parameters of the fully developed branch-and-bound tree (for example, tree-width, etc.) which are not known in advance. We plan to devise a branching strategy that ensures existence of sufficient number of nodes in every iteration, and is unambiguous, hence, ensures that $T(k_2) \leq T(k_1),\ k_2 > k_1 > 1$.

**Table 3.8:** Instances for which *mcbnbDeter* does not always improve when using more processors in terms of wall clock times when guided diving is disabled.

| name | *mcbnbDeter1* | *mcbnbDeter2* | *mcbnbDeter4* | *mcbnbDeter8* | *mcbnbDeter16* |
|---|---|---|---|---|---|
| batch0812 | 10.82 | 18.79 | 20.56 | 15.79 | 21.54 |
| clay0203h | 252.85 | 303.92 | 298.48 | 136.21 | 170.09 |
| clay0303h | 213.87 | 221.37 | 241.81 | 190.84 | 177.63 |
| clay0304h | 1284.48 | 782.48 | 2474.29 | 2717.52 | 2085.73 |
| cvxnonsep_normcon40 | 13.83 | 8.51 | 7.88 | 8.25 | 15.37 |
| cvxnonsep_normcon40r | 6.48 | 4.05 | 4.27 | 3.93 | 8.53 |
| cvxnonsep_pcon40r | 5.07 | 4.1 | 3.47 | 3.11 | 5.71 |
| du-opt5 | 7.29 | 6.8 | 7.74 | 11.12 | 16.02 |
| du-opt | 7.64 | 7.1 | 10.25 | 14.72 | 29.69 |
| enpro48pb | 23.17 | 25.31 | 26.34 | 34.81 | 71.29 |
| netmod_kar2 | 676.65 | 376.77 | 954.1 | 359.25 | 536.67 |
| portfol_roundlot | 66.09 | 115.87 | 28.77 | 15.29 | 21.54 |
| ravempb | 7.53 | 7.35 | 11.31 | 8.99 | 11.08 |
| rsyn0805m02h | 73.09 | 176.17 | 252.18 | 205.77 | 385.07 |
| rsyn0805m03h | 304.05 | 318.51 | 356.52 | 534.59 | 791.87 |
| rsyn0805m04h | 257.64 | 441.85 | 756.56 | 490.87 | 1196.29 |
| rsyn0810h | 6.1 | 6.47 | 11.46 | 15.13 | 30.21 |
| rsyn0810m03h | 1341.81 | 1730.02 | 1535.02 | 1429.5 | 1557.24 |
| rsyn0810m04h | 801.26 | 921.04 | 716.32 | 994.85 | 1518.04 |
| rsyn0815h | 9.13 | 9.71 | 16.28 | 20.86 | 40.46 |
| rsyn0815m02h | 231.65 | 316.94 | 423.33 | 290.99 | 470.15 |
| rsyn0815m04h | 1434.36 | 1144.03 | 1112.83 | 1414.65 | 1951.18 |
| rsyn0820h | 10.62 | 10.95 | 17.9 | 21.95 | 43.32 |
| rsyn0830h | 9.02 | 9.63 | 15.71 | 21.14 | 37.76 |
| rsyn0830m02h | 328.46 | 425.25 | 534.07 | 522.54 | 820.07 |
| rsyn0830m03h | 1536.98 | 1058.36 | 1172.37 | 1234.5 | 1654.07 |
| rsyn0840h | 5.54 | 8.56 | 16.49 | 23.76 | 44.2 |
| rsyn0840m02h | 182.32 | 640.07 | 462.22 | 468.44 | 698.54 |
| rsyn0840m03h | 1873.45 | 2172.79 | 1532.92 | 1330.83 | 1983.27 |
| slay05h | 9.83 | 11.83 | 19.63 | 23.44 | 38.73 |
| slay06h | 41.39 | 36.6 | 46.08 | 54.77 | 87.15 |
| slay06m | 12.18 | 13.13 | 23.39 | 24.45 | 44.74 |
| slay07h | 22.55 | 46.74 | 72.82 | 93.4 | 151.03 |
| slay07m | 9.49 | 15.18 | 33.68 | 40.74 | 74.73 |
| smallinvDAXr1b020-022 | 7.9 | 6.08 | 5.07 | 5.67 | 10.35 |
| smallinvDAXr2b020-022 | 7.89 | 5.88 | 4.75 | 5.19 | 9.9 |
| smallinvDAXr3b020-022 | 7.9 | 5.95 | 5.32 | 5.43 | 10.47 |
| smallinvDAXr4b020-022 | 7.88 | 5.9 | 5.44 | 5.65 | 10.47 |
| smallinvDAXr5b020-022 | 8.04 | 5.85 | 5.18 | 5.77 | 10.3 |
| squfl010-040 | 5.46 | 4.7 | 5.99 | 4.96 | 5.17 |
| syn15m02m | 24.66 | 22.19 | 22.85 | 23.22 | 37.56 |
| syn20m04h | 8.41 | 14.12 | 16.28 | 17.91 | 19.41 |
| syn30m03h | 10.39 | 13.55 | 19.22 | 20.94 | 22.59 |
| syn30m04h | 26.6 | 37.34 | 56.11 | 64.61 | 72.83 |
| syn30m | 15.79 | 10.07 | 10.52 | 9.55 | 24.77 |
| syn40m03h | 28.05 | 37.71 | 69.68 | 74.78 | 124.58 |
| syn40m04h | 131.84 | 109.87 | 134.37 | 159.15 | 237.74 |
| syn40m | 513.39 | 2279.12 | 2045.22 | 769.8 | 508.7 |
| watercontamination0202 | 30.7 | 24.99 | 31.5 | 24.38 | 25.87 |

**Table 3.9:** (Top) Comparison of opportunistic *mcqg1* with opportunistic *mcqg* using multiple threads. *mcqg1* could solve 320 instances. (Bottom) Break-up of performance of *mcqg16* over instances of varying difficulty.

| # threads | # solved by | | wall time | | nodes | |
|---|---|---|---|---|---|---|
| (T) | *mcqgT* | both | *mcqg1* | rel. | *mcqg1* | rel. |
| 2 | 320 | 317 | 16.94 | 0.84 | 1556.07 | 1.16 |
| 4 | 331 | 319 | 17.75 | 0.58 | 1632.70 | 1.13 |
| 8 | 333 | 319 | 17.75 | 0.42 | 1632.70 | 1.18 |
| 16 | 336 | 317 | 17.35 | 0.35 | 1584.00 | 1.32 |

| | # solved | wall time | | nodes | |
|---|---|---|---|---|---|
| time | by both | *mcqg1* | rel. | *mcqg1* | rel. |
| > 0 | 317 | 17.35 | 0.35 | 1584.00 | 1.32 |
| > 10 | 122 | 100.09 | 0.21 | 29631.78 | 1.37 |
| > 100 | 49 | 430.60 | 0.15 | 108404.43 | 1.19 |
| > 500 | 21 | 1342.86 | 0.12 | 197118.48 | 1.33 |

**Table 3.10:** Instances for which the wall clock time taken by *mcqgOppor* does not always improve when more processors are used.

| name | *mcqgOppor1* | *mcqgOppor2* | *mcqgOppor4* | *mcqgOppor8* | *mcqgOppor16* |
|---|---|---|---|---|---|
| ball_mk4_05 | 5.69 | 9.65 | 43.47 | 24.73 | 409.35 |
| ball_mk4_10 | 18.12 | 3600 | 3600 | 7200 | 7200 |
| ball_mk4_15 | 3600 | 3600 | 696.64 | 15.74 | 7200 |
| clay0203h | 41.61 | 26.91 | 67.49 | 74.6 | 53.87 |
| clay0205h | 19.07 | 20.14 | 38.77 | 4.56 | 4.4 |
| clay0304h | 194.18 | 238.08 | 127.37 | 43.39 | 86.9 |
| clay0305h | 118.21 | 101.05 | 124.2 | 47.96 | 45.84 |
| color_lab6b_4x20 | 641.89 | 625.97 | 777.15 | 749.55 | 833.04 |
| cvxnonsep_pcon40 | 66.09 | 37.46 | 22.07 | 14.43 | 7200 |
| cvxnonsep_psig20r | 0.85 | 0.79 | 1.52 | 12.26 | 4.36 |
| cvxnonsep_psig40r | 1.93 | 1.47 | 0.98 | 0.8 | 12.42 |
| fo7_ar2_1 | 76.75 | 84.91 | 32.81 | 21.78 | 15.42 |
| fo7_ar4_1 | 60.51 | 86.87 | 33.77 | 14.33 | 8.91 |
| fo8_ar3_1 | 110.22 | 230.94 | 68.32 | 17.43 | 9.51 |
| fo8_ar4_1 | 127.54 | 208.77 | 40.95 | 37.82 | 27.21 |
| fo8 | 2871.04 | 3600 | 521.47 | 355.3 | 215.84 |
| fo9_ar3_1 | 435.79 | 883.13 | 80.68 | 64.77 | 69.9 |
| fo9_ar4_1 | 609 | 1513.52 | 254.66 | 99.67 | 173.5 |
| fo9_ar5_1 | 3118.45 | 3600 | 422.37 | 241.14 | 133.74 |
| jbearing100 | 8.92 | 9.42 | 11.12 | 14.03 | 16.85 |
| jbearing50 | 3.25 | 3.34 | 4.22 | 5.41 | 6.92 |
| jbearing75 | 5.84 | 6.15 | 7.29 | 9.04 | 11.51 |
| m7_ar5_1 | 14.31 | 26.08 | 8.79 | 3.21 | 4.94 |
| netmod_dol2 | 1321.21 | 3435.2 | 359.61 | 392.7 | 111.03 |
| netmod_kar1 | 24.61 | 144.48 | 13.34 | 4.97 | 5.4 |
| netmod_kar2 | 24.72 | 123.96 | 12.94 | 6.01 | 3.73 |
| o7_ar3_1 | 3091.33 | 3140.58 | 1250.44 | 532.52 | 280.45 |
| rsyn0810m02m | 18.99 | 19.68 | 7.71 | 4.56 | 4.52 |
| rsyn0810m04h | 6.37 | 6.5 | 5.36 | 5.26 | 5.9 |
| rsyn0815m04m | 587.61 | 306.54 | 1033.32 | 182.84 | 87.95 |
| rsyn0820m04m | 456.17 | 1241.64 | 414.35 | 210.35 | 152.04 |
| rsyn0830m02m | 20.53 | 126.52 | 10.8 | 10.32 | 6.84 |
| rsyn0840m02m | 70.62 | 430.8 | 59.64 | 10.15 | 7.67 |
| rsyn0840m04h | 24.26 | 27.73 | 10.72 | 9.52 | 9.01 |
| slay08m | 67 | 16.03 | 8.93 | 3.6 | 3.11 |
| slay09h | 258.65 | 306.46 | 218.16 | 77.39 | 79.93 |
| slay09m | 23.69 | 67.68 | 24.72 | 20.99 | 11.66 |
| squfl020-050 | 3600 | 3600 | 3600 | 7200 | 3600 |
| sssd15-06 | 512.44 | 619.87 | 230.69 | 125.75 | 72.3 |
| sssd16-07 | 922.6 | 442.28 | 263.17 | 357.15 | 7200 |
| watercontamination0202 | 22.1 | 20.14 | 23.01 | 17.98 | 14.57 |

**Table 3.11:** (Top) Comparison of deterministic *mcqg1* with deterministic *mcqg* using multiple threads. *mcqg1* could solve 221 instances. (Bottom) Break-up of performance of *mcqg16* over instances of varying difficulty.

| # threads | # solved by | | wall time | | nodes | | iters | |
|---|---|---|---|---|---|---|---|---|
| (T) | *mcqgT* | both | *mcqg1* | rel. | *mcqg1* | rel. | | |
| 2 | 226 | 221 | 24.47 | 0.85 | 854.01 | 1.03 | 854.01 | 0.70 |
| 4 | 228 | 220 | 24.32 | 0.75 | 829.97 | 1.04 | 829.97 | 0.49 |
| 8 | 231 | 220 | 24.32 | 0.70 | 829.97 | 1.07 | 829.97 | 0.36 |
| 16 | 238 | 221 | 24.47 | 0.57 | 854.01 | 1.11 | 854.01 | 0.26 |

| | # solved | wall time | | nodes | | iters | |
|---|---|---|---|---|---|---|---|
| time | by both | *mcqg1* | rel. | *mcqg1* | rel. | *mcqg1* | rel. |
| > 0 | 221 | 24.47 | 0.57 | 854.01 | 1.11 | 854.01 | 0.26 |
| > 10 | 91 | 169.92 | 0.39 | 12194.71 | 1.17 | 12194.71 | 0.11 |
| > 100 | 47 | 682.65 | 0.32 | 54414.81 | 1.22 | 54414.81 | 0.09 |
| > 500 | 27 | 1503.56 | 0.36 | 125115.19 | 1.32 | 125115.19 | 0.10 |

**Table 3.12:** Instances for which *mcqgDeter* shows anomalous behaviour in number of iterations.

| name | *mcqgDeter1* | *mcqgDeter2* | *mcqgDeter4* | *mcqgDeter8* | *mcqgDeter16* |
|---|---|---|---|---|---|
| `color_lab2_4x0` | 97 | 162 | 80 | 84 | 80 |
| `rsyn0815m02h` | 1816 | 2116 | 1083 | 548 | 290 |
| `rsyn0820m04h` | 3808 | 17032 | 1151 | 642 | 369 |
| `rsyn0830m03h` | 1107 | 1346 | 684 | 355 | 187 |
| `rsyn0830m04h` | 3142 | 4552 | 1644 | 600 | 481 |

# Chapter 4

# A Parallel Branch-and-Estimate Heuristic for Nonconvex MINLP

Most deterministic methods to solve nonconvex MINLPs are also based on branch-and-bound framework. A good survey of algorithms for nonconvex MINLP is given by **?** and **?**.

If (P) is a convex MINLP, we can obtain a tractable relaxation by merely ignoring the integrality constraints. The nonlinear programming (NLP) relaxation obtained in this way can be solved using techniques of nonlinear optimization (see, for instance, **???**).

When (P) is not a convex MINLP, the above-mentioned NLP techniques do not provide an optimal solution to the nonconvex NLP relaxation. Instead, they converge to a Karush-Kuhn-Tucker (KKT) point which may or may not be an optimal solution to the relaxation. Furthermore, the KKT point to which a method converges may depend on initial point from where the method commenced, and on other steps of the method. Therefore, a different relaxation must be considered for nonconvex MINLPs. When the problem (P) is *factorable*, we can create its linear relaxation (e.g. **???**) by first adding auxiliary variables, rewriting the constraints as a set of many more constraints each having elementary nonlinear functions, and then creating their linear under and over estimators. We can also create quadratic relaxations (**?**) and, sometimes, semidefinite relaxations (**?**) of such a problem (P). However, a relaxation constructed using these ways is usually quite weak in the sense that its optimal solution value may be much lower than that of (P), and one may have to explore many subproblems in a branch-and-bound tree to find an optimal solution.

Given a nonconvex problem (P), we propose a multi-start method to obtain good bounds on its nonlinear continuous relaxation by repeatedly solving the relaxation using different initial points. In order to speed up the solving process, we create several copies of the relaxation. These copies of the relaxation are solved in parallel, each starting from a different starting point. The best solution from these runs is chosen as an estimate for the optimal solution. Since it is difficult to ascertain whether a KKT point is the global optimal solution of a relaxation, our method is not guaranteed to reach the optimal solution. Its accuracy depends on the choice of the initial points and the number of times a relaxation is solved. We present five different schemes of selecting the initial point in Section 4.2. Our computational experiments show that the method is much faster than the exact methods and yields near-optimal solutions for most of the instances. Moreover, the method can be used for MINLPs that are not factorable but have twice continuously differentiable functions.

While multi-start heuristics like ours have been proposed earlier for solving continuous NLPs (**????**), to the best of our knowledge, ours is the first that exploits parallelism in a branch-and-bound framework while employing multi-start heuristics for solving MINLPs. We present the results of our computational experiments with benchmark problem instances in Section 4.3 and present our conclusions in Section 4.4.

## 4.1 The Branch-and-Estimate Heuristic

The main motivation for our method, termed as the Branch-and-Estimate Heuristic, comes from the NLP based branch-and-bound (NLP-BB) method for solving MINLPs. The natural branching scheme in this algorithm is to branch on integer-constrained variables, and we use the same branching scheme in our method. We say that a variable $x_k$ is a branching candidate if $k \in I$, and $\hat{x}_k \notin \mathbb{Z}$, where $\hat{x}$ is the optimal solution of the relaxation. For a given branching candidate $x_k$ for some $k \in I$, we can create two subproblems based on the disjunction $x_k \geq \lceil \hat{x}_k \rceil \vee x_k \leq \lfloor \hat{x}_k \rfloor$. Thus, a subproblem in the NLP-BB technique differs from the original problem only in the bounds of its variables. A subproblem can be represented by $NLP(l, u)$ where $l, u \in \mathbb{R}^n$ denote the lower and upper bounds respectively on the variables of a subproblem.

Our heuristic starts by presolving the given MINLP and then obtaining a relaxation of the presolved problem by removing the integer constraints on integer

constrained variables $x_k, \forall k \in I$. Since the resulting relaxation is not convex in general, an NLP solver may end up at a KKT point which may not be an optimal solution to the relaxation. In an attempt to find an optimal solution, we generate multiple initial points and restart the NLP solver from each of these points. This process can be readily performed in parallel since each solution process is independent of the others. We invoke one NLP solver routine at each available computational core or thread simultaneously, each solving the same NLP from a different initial point. We try to pick initial points in disjoint regions within the variables' bounds and pass it to each thread so that the NLP solver may converge to as many different KKT points as possible. Additionally, for each thread, we restart the NLP solver a fixed number of times by updating the initial point every time but within the same region allocated to that thread. From a thread, we finally obtain a 'thread-best' solution if at least one of the calls to solve the NLP returns a solution. We then take the best solution among those obtained from all the threads and regard it as the global solution of the continuous non-convex NLP (although there is no guarantee that this point would be the global solution). Then, as in branch-and-bound method, this subproblem is pruned if no thread finds a solution or if the best solution has a value greater than the available upper bound. If the best solution satisfies integer constraints, the upper bound is updated and the subproblem is discarded. Otherwise, we branch on an integer variable and get two more subproblems. The algorithm continues until all the subproblems are either pruned or solved.

A pseudocode of this method is provided in Algorithm 4.1. $T$ denotes the number of threads and $M$ is the number of starts per thread. These parameters are fixed according to the problem or user preferences. $\hat{x}_t^{(l,u)}$ denotes the best solution obtained by thread $t, t = 1, \ldots, T$ among those obtained from all $M$ calls to the NLP solver that it makes for a subproblem, and $TBestVal_t$ is the corresponding objective function value. $\hat{x}^{(l,u)}$ denotes the overall best solution among those obtained from all the threads, and $BestVal$, its objective function value. Lines 7 – 18 in the pseudocode are run in parallel on $T$ different threads, while the remaining lines are run on a single master thread. The initial point in Line 9 is obtained using one of the schemes explained in Section 4.2.

**Algorithm 4.1:** A Branch-and-Estimate heuristic for nonconvex MINLP

Set $U = \infty$ and initialize the list of open problems $\mathcal{H} = \emptyset$.
Add $NLP(-\infty, \infty)$ to the list: $\mathcal{H} \cup NLP(-\infty, \infty)$.
**while** $\mathcal{H} \neq \emptyset$ **do**
    Remove a problem $NLP(l, u)$ from the list $\mathcal{H}$.
    Initialize $BestVal = \infty$.
    Initialize $TBestVal_t = \infty,\ t = 1, \ldots, T$.
    **for** *t from* 1 *to* $T$ **do**
        **for** *j from* 1 *to* $M$ **do**
            Pick an initial point $x_{tj}^{(l,u)}$.
            Solve $NLP(l, u)$ starting from $x_{tj}^{(l,u)}$.
            **if** *solver returns a feasible* $\bar{x}_{tj}^{(l,u)}$ **then**
                **if** $f(\bar{x}_{tj}^{(l,u)}) < TBestVal_t$ **then**
                    $TBestVal_t = f(\bar{x}_{tj}^{(l,u)})$.
                    Set $\hat{x}_t^{(l,u)} = \bar{x}_{tj}^{(l,u)}$.
    $BestVal = \min_t\{TBestVal_t, t = 1, \ldots, T\}$.
    $\hat{x}^{(l,u)} = argmin_t\{f(\hat{x}_t^{(l,u)}) \mid TBestVal_t < \infty\}$.
    **if** $BestVal = \infty$ **then**
        $NLP(l, u)$ is assumed infeasible and pruned.
    **else if** $f(\hat{x}^{(l,u)}) > U$ **then**
        $NLP(l, u)$ can be pruned on the basis of bound.
    **else if** $\hat{x}_i^{(l,u)} \in \mathbb{Z},\ \forall\ i \in I$ **then**
        Update the incumbent solution:
        $$U = f(\hat{x}^{(l,u)}), x^* = \hat{x}^{(l,u)}.$$
    **else**
        Use an appropriate branching rule to generate $NLP(l^-, u^-)$ and $NLP(l^+, u^+)$.
        Update the list: $\mathcal{H} = \mathcal{H} \cup \{NLP(l^-, u^-), NLP(l^+, u^+)\}$.

## 4.2 Initial Point Generation Schemes

Selecting an initial point closer to a global optimal solution of a nonconvex nonlinear program may make the NLP solver converge to the global solution, as most

of the NLP algorithms iteratively make use of information available about the *local* neighborhood of an iterate. This property motivated the constraint consensus (CC) method (**???**) which tries to obtain a near-feasible solution of a nonconvex NLP.

The goal of our method is to solve the global optimization problem, hence, we want to search the feasible space efficiently. By starting from different initial points that are generated in disjoint regions of the variable space, we hope to converge to distinct KKT points, some of which might be globally optimal or near optimal. For generating potentially good initial points, we propose five different 'randomized' schemes. We call the schemes randomized because we use random number generators within the schemes. Some of these schemes dynamically use the information generated after an NLP solver runs, e.g. the previous optimal point, the previous convergence direction, the distance of the optimal from the initial point, etc., while others are more randomized so as to encourage diversity of the KKT points found.

Before discussing the schemes, we introduce some common notation used for all the schemes to make our presentation clearer. $V$ is used to denote the set of all variables of the problem, and $ub_i$ and $lb_i$ denote the upper and lower bound on variable $i$, respectively. $T$ denotes the number of parallel threads spawned on the computing machine, and $M$ the number of different starts per thread. The number of restarts will therefore be $M - 1$. A subproblem (node) is solved from different initial points $T \times M$ number of times, on $T$ processors running in parallel. We denote the $i^{th}$ coordinate of the initial point vector generated in the $j^{th}$ start at thread $t$ by $x_{tji}^{(l,u)}$ and the most recent finite optimal objective function value obtained by solving $NLP(l,u)$, starting from some $x_{tj}^{(l,u)}$, as $z_t^{(l,u)}$ where $i \in V$, $j \in \{1,\ldots,M\}$ and $t \in \{1,\ldots,T\}$. We represent by $B$, the set of all indices $i \in \{1,\ldots,n\}$, such that both $lb_i$ and $ub_i$ are finite.

Our schemes generate points randomly using a function denote by RAND$(a,b)$ to generate a uniformly distributed random variable in $[a,b]$. Further, if $a$ is $-\infty$, the function generates a random number in the interval $[b-K,b]$, where the parameter $K$ is a fixed large integer value. Similarly, if $b$ is $\infty$, the function generates a random number in the interval $[a,a+K]$. If a variable is not bounded on either sides, it generates a random number in the interval $[0,K]$. The function RAND$(\{a,b\})$ denotes a function to randomly pick either $a$ or $b$ with equal probability.

### 4.2.1 Scheme-1

In this simple scheme, an initial point is randomly generated such that each of its coordinates lies within the the bounds of the corresponding variable. Algorithm 4.2 presents a pseudocode for this scheme.

**Algorithm 4.2:** Scheme-1 for initial point generation for a given $t \in \{1, \ldots, T\}$.

1 **for** *j from* 1 *to* $M$ **do**
2 Set $x_{tji}^{(l,u)} = \text{RAND}(lb_i, ub_i)$.

### 4.2.2 Scheme-2

In this scheme, we first find a bounded variable $x_k, k \in \{1, 2, \ldots, n\}$ whose upper and lower bounds are the farthest apart but finite. Then we create $T$ equal partitions of the interval $[lb_k, ub_k]$ and randomly generate the $k^{th}$ component of the initial point vector in these partitions respectively for each thread. Rest of the components are generated randomly as in Scheme-1. This logic gives us the first initial point for each thread for each subproblem, as shown in Algorithm 4.4.

We generate the starting point for the subsequent NLP calls by taking a random unit direction and finding a point outside a region of a fixed radius $r$, centered around the previous starting point. If consecutive starting points result in the same objective function value, we scale the radius $r$ with a parameter, $s$, for generating the next starting point. Otherwise, the same radius is used in the next round as well. A pseudocode for this scheme is shown in Algorithm 4.3.

**Algorithm 4.3:** Scheme-2 for initial point generation for a given $t \in \{1, \ldots, T\}$ and $j \in \{2, \ldots, M\}$.

1 Generate a random unit vector $a_{tj}$ of dimension $n$.
2 $x_{tji}^{(l,u)} = max(min(x_{t(j-1)i}^{(l,u)} + r \cdot a_{tji}, ub_i), lb_i), \forall i \in V$.
3 **if** *NLP*$(l, u)$ *returns the same value as in previous trial* **then**
4 $r = r \cdot s$.

### 4.2.3 Scheme-3

This scheme differs from Scheme-2 in the following two ways. First, instead of using a fixed value of $r$, we change it on the basis of the distance between

**Algorithm 4.4:** Initial point generation for $j = 1$ for a given $t \in \{1, \ldots, T\}$ in Scheme-2.

1 $x_{tji}^{(l,u)} = \text{RAND}(lb_i, ub_i), \forall i \in V.$
2 **if** $B \neq \emptyset$ **then**
3 Find a $k \in B$: $ub_k - lb_k \geq ub_i - lb_i, \forall i \in B.$
4 Set $q = ub_k - lb_k.$
5 $x_{tjk}^{(l,u)} = lb_k + \text{RAND}((t-1)\frac{q}{T}, t\frac{q}{T}).$

the previous starting point and the corresponding previous optimal solution obtained. Second, instead of the previous starting point, we take the previous optimal solution as the center. The scaling parameter, $s$, is used as in Scheme-2.

### 4.2.4 Scheme-4

This scheme is similar to Scheme-3 except that when we randomly generate a unit vector for selecting the next starting point, we accept it only if it lies outside a cone of 'unfavourable' directions. We consider this cone as the set of all directions which make a small angle with the line joining the previous starting point and the previous optimal solution. A parameter for the threshold angle, $\theta$, is used. We keep generating random directions until we obtain one that points outside this cone. We choose the radius in this scheme similar to Scheme-3, as a scaled distance between the previous starting point and the corresponding previous optimal solution.

### 4.2.5 Scheme-5

Here, the first initial point for each thread is generated randomly within the bounds of the variables in a separate thread-specific region in a different way than the other schemes. If both bounds of a variable are available, then either the upper bound or the lower bound is chosen randomly as the initial value for that variable. Otherwise, a random value is generated within the available bounds. We refer to such a point as a 'box corner'. In the next iteration, we use the previous optimal solution and locate a box corner which is farthest from this point. Then, along the direction of this farthest box corner starting from the previous optimal solution, we take a radius as in Scheme-3 and locate a temporary point. Finally, we take a random convex combination of this temporary point and the

farthest box corner as the next initial point. A pseudocode for this scheme is shown in Algorithm 4.5.

**Algorithm 4.5:** Scheme-5 for initial point generation for a given $t \in \{1,\ldots,T\}$ and a $j \in \{1,\ldots,M\}$.

**if** *no solution has been found yet in thread* $t$ **then**
  Find a random box corner $x_{tj}^{(l,u)}$ as per Algorithm 4.6.
**else**
  Find the farthest box corner $w_{tj}^{(l,u)}$ as per Algorithm 4.7.
  Find a temporary point outside a radius $r$: let $d_{tj}$ be the unit vector parallel to $w_{tj}^{(l,u)} - \bar{x}_{t(j-1)}^{(l,u)}$.
  Set $r = \|\bar{x}_{t(j-1)}^{(l,u)} - x_{t(j-1)}^{(l,u)}\|$.
  Set $v_{tj}^{(l,u)} = \bar{x}_{t(j-1)}^{(l,u)} + r \cdot d_{tj}$.
  Take a random convex combination of $v_{tj}$ and $w_{tji}^{(l,u)}$ as the new initial point.

**Algorithm 4.6:** Generating a random box corner for a given $j \in \{1,\ldots,M\}$ and a $t \in \{1,\ldots,T\}$ for Scheme-5.

**if** $B \neq \emptyset$ **then**
  Find a $k \in B$: $ub_k - lb_k \geq ub_i - lb_i, \forall i \in B$ Set $q = ub_k - lb_k$ $x_{tjk}^{(l,u)} = lb_k +$ RAND($\{(t-1)\frac{q}{T}, t\frac{q}{T}\}$) $x_{tji}^{(l,u)} =$ RAND($\{lb_i, ub_i\}$), $\forall i \in B, i \neq k$
  $x_{tji}^{(l,u)} =$ RAND($lb_i, ub_i$), $\forall i \in V, i \notin B$
**else**
  $x_{tji}^{(l,u)} =$ RAND($lb_i, ub_i$), $\forall i \in V$

## 4.3 Computational Results

We compare the results obtained from our multi-start heuristic using the above schemes with the NLP-BB algorithm without restarts (which is also a heuristic for nonconvex MINLP) and two global solvers SCIP (**?**) and Couenne (**?**). Both these global solvers use linear underestimators and overestimators for obtaining the convex relaxations.

**Algorithm 4.7:** Generating the farthest box corner from a given point $\bar{x}_{tj}^{(l,u)}$, $j \in \{1,\ldots,M\}$ and a $t \in \{1,\ldots,T\}$ for Scheme-5.

**if** $B \neq \emptyset$ **then**
  Find a $k \in B$: $ub_k - lb_k \geq ub_i - lb_i, \forall i \in B$.
  Define $q = ub_k - lb_k$.
  **if** $(\bar{x}_{tji}^{(l,u)} \geq 0.5(lb_i + ub_i))$ **then**
    $x_{tjk}^{(l,u)} = lb_k + (t-1)\frac{q}{T}$.
    $x_{tji}^{(l,u)} = lb_i, \forall i \in B, i \neq k$.
  **else**
    $x_{tjk}^{(l,u)} = lb_k + t\frac{q}{T}$.
    $x_{tji}^{(l,u)} = ub_i, \forall i \in B, i \neq k$.
  $x_{tji}^{(l,u)} = \text{RAND}(lb_i, ub_i), \forall i \in V, i \notin B$.
**else**
  $x_{tji}^{(l,u)} = \text{RAND}(lb_i, ub_i), \forall i \in V$.

### 4.3.1 Experimental Setup

We selected nonconvex problems from the benchmark set MINLPLib (**?**). We ran all our tests on a system described in Section 2.2. The NLP-BB method (without restarts), the multi-start heuristic and all the initial point generation schemes were implemented within Minotaur (**?**). All codes were compiled with GCC4.7.2 and OpenMP3.1. IPOPT-3.11.9, with MA27 linear-systems solver was used as the NLP solver. We set 'presolve=true' for all the test problems and used the 'Maximum violation' brancher in Minotaur. IPOPT was used as the sole NLP solver. For the multi-start heuristic, we used 10 threads with 4 starts each. All other algorithms were run on a single thread. The wall clock time limit for all the algorithms was set to 1800 seconds. For Scheme-2, $r$ was set to 0.8. Parameter $s$ was set to 1.5 in schemes 2, 3 and 4. For Scheme-4, we set the threshold angle, $\theta$, to 30 degrees. $K$ was set to 999 in all schemes.

### 4.3.2 Inferences

We denote the NLP-BB algorithm by *bnb*, our multi-start heuristic by *msbnb*, SCIP by *scip* and Couenne by *couenne* in forthcoming text and tables. To refer to different initial point generation schemes, an $S$ followed by its identifier number $(1-5)$, is appended to *msbnb*. For comparing the performance of all

schemes with *bnb* and the global solvers, we ran our heuristic on 101 problems. Table 4.1 and Table 4.2 provide detailed results of our runs. The instance names along with best known values of primal and dual bounds are shown. Except the last 3 problems in Table 4.3, all others are of minimization type. The results for *bnb*, *scip* and *couenne* are also shown. $z^*$ denotes the best objective function value reached by an algorithm, and the column *time* gives the 'wall clock time' in seconds. A '–' or 'inf' indicates that a feasible solution could not be obtained, or the solver stopped because of some error.

We observe that *msbnb* with any scheme for generating initial points gives better solutions in most of the instances than *bnb*. In particular, *msbnbS1* reports a better solution for more than 85 instances as compared to *bnb*. There does not seem to be much overall difference in the performance of different *msbnb* schemes, but they tend to have significant differences on some instances. For example, *msbnbS5* and *msbnbS4* both give same solutions for 45 problems, the former performs better in 24 instances and the latter in 24 cases. Overall, the average percentage gap between the best known primal bound and $z^*$ taken over all the instances, for all *msbnb* schemes is less than 5%, with a maximum gap of 200%. It shows that on the whole, the *msbnb* is working reasonably well, even though there are quite a few instances where the solution obtained is far from the optimal.

We have used performance profiles (**?**) for benchmarking *msbnb* schemes with *bnb*. The performance profiles in Figure 4.1 show that all the *msbnb* heuristics solve about 50% of the instances within a gap of 0.05% of the best known primal bound, while *bnb* solves just one (`shiporig`), showing that it can not be the choice when reasonable solutions are desired. Figure 4.2 shows a similar plot for 1% optimality gap, demonstrating that except about 20% of the instances, all *msbnb* schemes could reach within a good proximity of the best known optimal solution. In Figure 4.3, we show the performance profiles when 10% optimality gap is acceptable. Here, *bnb* and *msbnb* schemes gave acceptable solutions for about 50% and 80% of the instances respectively. In Figure 4.4, we omit the restriction on the quality of the solution reached by the algorithms and observe that *bnb* is the fastest, which is not surprising because it solves the NLP relaxation at each node just once, while all the *msbnb* heuristics solve them $T \times M$ number of times. Figure 4.5 shows the performance profiles based on the number of nodes created in the branch-and-bound tree. The number of nodes is roughly the same for all *msbnb* variants and *bnb* for about 80% of the instances.

As compared to *scip*, *msbnbS5* heuristic gives better solutions in 57 cases while they are at par for 30 cases. For most of the problems where *msbnbS5* performed better, *scip* consumed its full quota of execution time (1800 seconds) while the former always took less than half of this time. The corresponding numbers for *couenne* are 39 and 46. For the odd instance `autocorr_bern50-13`, the heuristics *msbnbS1* and *msbnbS2* have hit a better solution than the best known primal bound within 16 seconds, while *scip* and *couenne* reach the time limit and report a solution with an inferior objective function value.

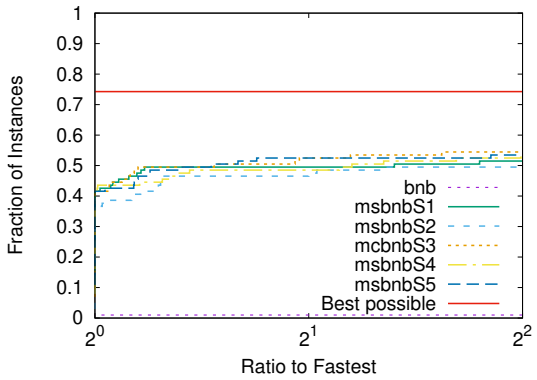



**Figure 4.1:** Performance profiles based on wall clock time for 101 nonconvex instances from MINLPLib (optimality gap 0.05%).

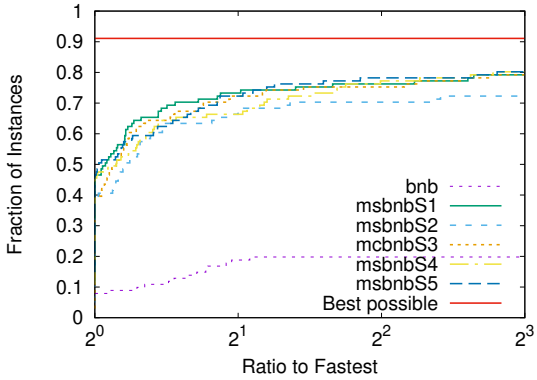



**Figure 4.2:** Performance profiles based on wall clock time for 101 nonconvex instances from MINLPLib (optimality gap 1%).

## 4.4 Conclusions and Future Research

We have implemented and evaluated a heuristic for solving nonconvex MINLPs in a branch-and-bound framework. We have also presented a few schemes that explore disjoint regions in the variable space and provide a variety of starting points to NLP solvers. Since solving relaxations is the most time taking step while solving a MINLP, we use available multiple cores in parallel to obtain several KKT points, and then select the best one among them.

Our computational experiments show that the proposed multi-start heuristic gives better quality solutions compared to the NLP-BB algorithm (without any restarts), which is fast but not accurate. Our heuristic has terminated before reaching the time limit for almost all the instances. Thus, if we solve a nonconvex problem using NLP restarts, much better solutions can be obtained.

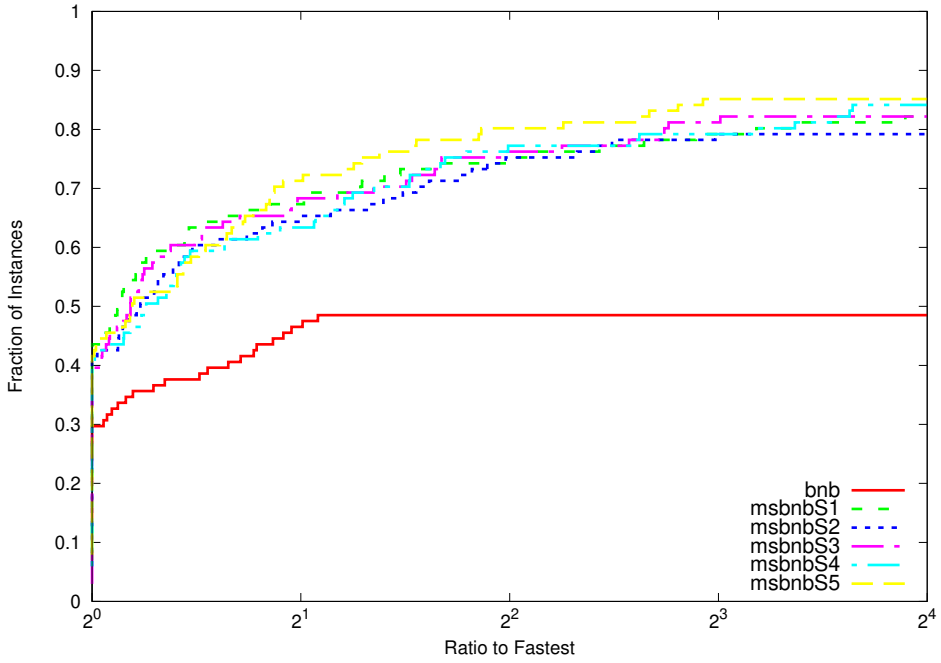



**Figure 4.3:** Performance profiles based on wall clock time for 101 nonconvex instances from MINLPLib (optimality gap 10%).

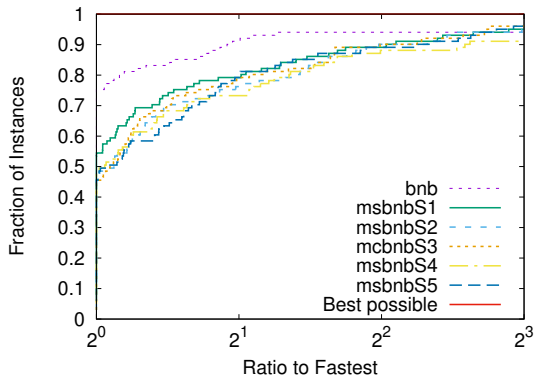



**Figure 4.4:** Performance profiles based on wall clock time for 101 nonconvex instances (without checking solution quality).

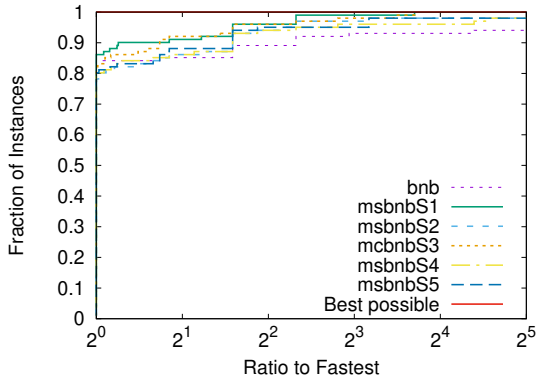



**Figure 4.5:** Performance profiles based on number of nodes created for 101 nonconvex instances.

More sophisticated initial point generation schemes for exploiting problem specific information of MINLPs can be developed for obtaining even better solutions in lesser time. To speed up the heuristic, one can exploit more parallelism and share more information among different threads and subproblems. It would be interesting to study how other MINLP heuristics (**????**) compare with ours and how one can incorporate their strengths into ours.

**Table 4.1:** Performance of schemes and solvers on MINLPLib instances (I)

| Name | Dual Bd | Primal Bd | bnb | | msbnbS1 | | msbnbS2 | | msbnbS3 | | msbnbS4 | | msbnbS5 | | scip | | couenne | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | $z^*$ | time | $z^*$ | time | $z^*$ | time | $z^*$ | time | $z^*$ | time | $z^*$ | time | $z^*$ | time | $z^*$ | time |
| arki0004 | -3109.97 | 323.47 | 327.73 | 3.56 | 323.47 | 36.74 | 323.47 | 94.37 | 323.47 | 84.06 | 323.47 | 36.71 | 327.73 | 391.23 | – | 1800 | 327.73 | 1800 |
| autocorr_bern20-05 | -416.00 | -416.00 | -408.00 | 0.08 | -412.00 | 0.89 | -412.00 | 1.82 | -416.00 | 0.44 | -412.00 | 2.22 | -412.00 | 0.75 | -416.00 | 1800 | -416.00 | 87.62 |
| autocorr_bern20-10 | -2936.00 | -2936.00 | -2912.00 | 1.2 | -2920.00 | 2.17 | -2920.00 | 1.57 | -2920.00 | 1.96 | -2920.00 | 3.44 | -2912.00 | 1.71 | -2904.00 | 1800 | -2936.00 | 250.62 |
| autocorr_bern20-15 | -5960.00 | -5960.00 | -5820.00 | 1.01 | -5944.00 | 2.09 | -5924.00 | 2.23 | -5916.00 | 2.28 | -5912.00 | 2.40 | -5944.00 | 1.8 | -5960.00 | 1800 | -5960.00 | 755.36 |
| autocorr_bern25-06 | -960.00 | -960.00 | -928.00 | 0.78 | -960.00 | 0.82 | -960.00 | 2.00 | -960.00 | 4.76 | -960.00 | 2.55 | -960.00 | 3.61 | -944.00 | 1800 | -960.00 | 1800 |
| autocorr_bern25-13 | -8888.00 | -8148.00 | -7912.00 | 1.34 | -8108.00 | 12.16 | -8060.00 | 3.93 | -8132.00 | 4.18 | -8136.00 | 3.85 | -8144.00 | 8.27 | -8040.00 | 1800 | -8148.00 | 1800 |
| autocorr_bern25-19 | -25502.58 | -14644.00 | -14412.00 | 4.4 | -14544.00 | 4.84 | -14596.00 | 5.87 | -14604.00 | 5 | -14644.00 | 5.90 | -14604.00 | 8.86 | -14600.00 | 1800 | -14588.00 | 1800 |
| autocorr_bern25-25 | -24157.48 | -10664.00 | -10360.00 | 6.4 | -10644.00 | 7.14 | -10652.00 | 6.09 | -10644.00 | 7.44 | -10640.00 | 6.34 | -10624.00 | 6.73 | -10528.00 | 1800 | -10620.00 | 1800 |
| autocorr_bern30-08 | -4519.33 | -2952.00 | -2904.00 | 1.34 | -2948.00 | 2.08 | -2952.00 | 2.24 | -2952.00 | 2.65 | -2952.00 | 1.80 | -2952.00 | 1.86 | -2872.00 | 1800 | -2952.00 | 1800 |
| autocorr_bern30-15 | -35794.65 | -15744.00 | -15464.00 | 3.08 | -15596.00 | 8.57 | -15568.00 | 9.43 | -15652.00 | 9.82 | -15720.00 | 7.12 | -15652.00 | 5.81 | -15484.00 | 1800 | -15660.00 | 1800 |
| autocorr_bern30-23 | -118159.66 | -30420.00 | -30236.00 | 14.12 | -30372.00 | 11.52 | -30364.00 | 15.97 | -30416.00 | 13.45 | -30424.00 | 13.55 | -30292.00 | 18.02 | -30012.00 | 1800 | -30208.00 | 1800 |
| autocorr_bern30-30 | -96115.45 | -22888.00 | -22376.00 | 17.1 | -22784.00 | 18.78 | -22808.00 | 17.11 | -22816.00 | 19.38 | -22888.00 | 15.70 | -22784.00 | 17.95 | -22200.00 | 1800 | -22664.00 | 1800 |
| autocorr_bern35-09 | -11522.16 | -5108.00 | -5040.00 | 1.29 | -5084.00 | 19.19 | -5068.00 | 3.62 | -5076.00 | 3.73 | -5096.00 | 3.90 | -5080.00 | 3.16 | -4860.00 | 1800 | -5068.00 | 1800 |
| autocorr_bern35-18 | -115172.32 | -31160.00 | -31072.00 | 15.23 | -31152.00 | 15.70 | -31096.00 | 16.86 | -31104.00 | 15.75 | -31152.00 | 15.44 | -31128.00 | 17.01 | -30320.00 | 1800 | -31072.00 | 1800 |
| autocorr_bern35-26 | -287095.39 | -55184.00 | -55112.00 | 53.48 | -55080.00 | 32.24 | -55136.00 | 34.17 | -55112.00 | 45.08 | -55056.00 | 34.72 | -55264.00 | 31.27 | -54000.00 | 1800 | -55032.00 | 1800 |
| autocorr_bern35-35 | -396664.00 | -41052.00 | -40564.00 | 68.08 | -41040.00 | 37.42 | -40636.00 | 33.85 | -41000.00 | 43.94 | -40908.00 | 39.88 | -40824.00 | 44.92 | -21388.00 | 1800 | -40284.00 | 1800 |
| autocorr_bern40-05 | -1868.00 | -936.00 | -912.00 | 0.26 | -928.00 | 2.02 | -924.00 | 3.53 | -920.00 | 3.19 | -924.00 | 2.10 | -924.00 | 3.64 | -872.00 | 1800 | -936.00 | 1800 |
| autocorr_bern40-10 | -23724.00 | -8232.00 | -8144.00 | 1.4 | -8176.00 | 6.01 | -8192.00 | 5.20 | -8184.00 | 5.56 | -8168.00 | 17.49 | -8184.00 | 10.64 | -7912.00 | 1800 | -8176.00 | 1800 |
| autocorr_bern40-20 | -235816.80 | -50516.00 | -49836.00 | 18.78 | -50384.00 | 25.55 | -50240.00 | 28.27 | -50404.00 | 26.41 | -50452.00 | 24.62 | -50488.00 | 24.9 | -49836.00 | 1800 | -50224.00 | 1800 |
| autocorr_bern40-30 | -705600.00 | -94768.00 | -94288.00 | 83.24 | -94672.00 | 58.27 | -94712.00 | 63.70 | -94848.00 | 61.82 | -94584.00 | 121.77 | -94760.00 | 59.03 | -91304.00 | 1800 | -94184.00 | 1800 |
| autocorr_bern45-05 | -2220.00 | -1068.00 | -1024.00 | 2.03 | -1044.00 | 8.31 | -1048.00 | 5.34 | -1048.00 | 5.76 | -1044.00 | 3.52 | -1060.00 | 5.27 | -1024.00 | 1800 | -1068.00 | 1800 |
| autocorr_bern45-11 | -45476.00 | -12660.00 | -12512.00 | 1.52 | -12628.00 | 8.19 | -12616.00 | 8.54 | -12648.00 | 10.03 | -12632.00 | 9.12 | -12568.00 | 4.46 | -9832.00 | 1800 | -12488.00 | 1800 |
| autocorr_bern45-23 | -581992.57 | -85200.00 | -84644.00 | 54.45 | -85216.00 | 35.56 | -85136.00 | 39.37 | -85188.00 | 38.83 | -85140.00 | 33.25 | -85276.00 | 60.09 | -83036.00 | 1800 | -84944.00 | 1800 |
| autocorr_bern45-34 | -1786641.40 | -152192.00 | -151912.00 | 168.31 | -151912.00 | 99.07 | -152344.00 | 103.44 | -152128.00 | 89.2 | -151904.00 | 118.03 | -151968.00 | 120.72 | -113440.00 | 1800 | -151648.00 | 1800 |
| autocorr_bern45-45 | -1817342.20 | -112764.00 | -112084.00 | 144.61 | -112908.00 | 98.57 | -112548.00 | 132.98 | -112456.00 | 111.97 | -112000.00 | 117.97 | -112912.00 | 111.79 | -102840.00 | 1800 | -110936.00 | 1800 |
| autocorr_bern50-06 | -6336.00 | -2160.00 | -2096.00 | 3.74 | -2160.00 | 8.77 | -2160.00 | 18.01 | -2144.00 | 24.06 | -2144.00 | 42.21 | -2112.00 | 3.6 | -2096.00 | 1800 | -2136.00 | 1800 |
| autocorr_bern50-13 | -102888.00 | -23544.00 | -23104.00 | 5.99 | -23576.00 | 14.68 | -23556.00 | 15.03 | -23392.00 | 48.22 | -23500.00 | 74.06 | -23524.00 | 10.8 | -22496.00 | 1800 | -23336.00 | 1800 |
| autocorr_bern50-25 | -1124105.47 | -124748.00 | -124016.00 | 46.58 | -124472.00 | 51.05 | -124612.00 | 64.99 | -124648.00 | 51.66 | -124708.00 | 52.86 | -124552.00 | 63.1 | -119852.00 | 1800 | -123892.00 | 1800 |
| autocorr_bern50-38 | -4131064.00 | -232496.00 | -231472.00 | 279.75 | -232568.00 | 188.10 | inf | 318.89 | -232496.00 | 309.92 | -232416.00 | 522.80 | -232440.00 | 162 | -174464.00 | 1800 | -231152.00 | 1800 |
| autocorr_bern55-06 | -8144.00 | -2400.00 | -2336.00 | 11.76 | -2400.00 | 6.45 | -2368.00 | 21.80 | -2392.00 | 43.73 | -2368.00 | 7.69 | -2384.00 | 15.38 | -2304.00 | 1800 | -2384.00 | 1800 |
| autocorr_bern55-14 | -181616.00 | -33144.00 | -32432.00 | 16.13 | -32832.00 | 119.75 | -32840.00 | 63.68 | -32704.00 | 24.9 | -33016.00 | 25.53 | -32888.00 | 77.11 | -30872.00 | 1800 | -32640.00 | 1800 |
| autocorr_bern55-28 | -2118883.17 | -190420.00 | -189720.00 | 143.83 | -190172.00 | 100.86 | -190228.00 | 74.20 | -190212.00 | 94.72 | -190012.00 | 95.05 | -190048.00 | 135.96 | -186196.00 | 1800 | -189084.00 | 1800 |
| autocorr_bern55-41 | — | -337370.00 | -335872.00 | 504.92 | -336916.00 | 275.75 | inf | 295.45 | -337540.00 | 282.24 | -336800.00 | 238.61 | -337120.00 | 407.49 | -299544.00 | 1800 | -3233218.00 | 1800 |

**Table 4.2:** Performance of schemes and solvers on MINLPLib instances (II)

| **Name** | **Dual Bd** | **Primal Bd** | **bnb** | | **msbnbS1** | | **msbnbS2** | | **msbnbS3** | | **msbnbS4** | | **msbnbS5** | | **scip** | | **couenne** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | $z^*$ | time | $z^*$ | time | $z^*$ | time | $z^*$ | time | $z^*$ | time | $z^*$ | time | $z^*$ | time | $z^*$ | time |
| autocorr_bern55-55 | — | -241912.00 | -239108.00 | 338.83 | -241380.00 | 408.02 | -241952.00 | 365.99 | -241604.00 | 305.64 | -242092.00 | 295.88 | -241696.00 | 301.65 | -189160.00 | 1800 | -2501535.00 | 1800 |
| autocorr_bern60-08 | -28132.00 | -6792.00 | -6648.00 | 5.18 | -6768.00 | 6.79 | -6768.00 | 6.08 | -6760.00 | 6.16 | -6768.00 | 8.04 | -6776.00 | 5.43 | -6452.00 | 1800 | -6744.00 | 1800 |
| autocorr_bern60-15 | -306778.00 | -44896.00 | -44144.00 | 11.43 | -44812.00 | 35.37 | -44640.00 | 90.51 | -44780.00 | 29.08 | -44728.00 | 39.63 | -44620.00 | 17.47 | -42584.00 | 1800 | -44408.00 | 1800 |
| autocorr_bern60-30 | -4486416.00 | -261046.22 | -258752.00 | 99.94 | -260448.00 | 132.70 | -260416.00 | 118.63 | -260272.00 | 93.6 | -260528.00 | 129.61 | -260576.00 | 214.89 | -203408.00 | 1800 | -259480.00 | 1800 |
| autocorr_bern60-45 | — | -478528.00 | -477304.00 | 674.77 | -478348.00 | 455.10 | inf | 444.75 | -479172.00 | 283.42 | -479000.00 | 433.22 | -479148.00 | 429.32 | -345468.00 | 1800 | -4620726.00 | 1800 |
| autocorr_bern60-60 | — | -350312.00 | -342924.00 | 501.16 | -350136.00 | 541.11 | -350516.00 | 557.68 | -349488.00 | 486.72 | -348700.00 | 498.18 | -350748.00 | 448.1 | – | – | -3641370.00 | 1800 |
| bayes2_50 | 0.00 | 0.52 | 62.41 | 0.1 | 0.52 | 0.46 | 0.52 | 0.43 | 0.52 | 0.41 | 0.52 | 0.42 | 0.52 | 0.45 | 5.67 | 1800 | 0.52 | 119.29 |
| edgecross10-060 | 459.00 | 459.00 | 463.00 | 0.92 | 459.00 | 1.65 | 459.00 | 1.76 | 459.00 | 1.44 | 459.00 | 3.18 | 459.00 | 1.65 | 459.00 | 86.43 | 459.00 | 173.98 |
| edgecross14-039 | 109.00 | 109.00 | 110.00 | 3.2 | 109.00 | 77.91 | 109.00 | 121.18 | 109.00 | 68.92 | 109.00 | 29.16 | 109.00 | 22.39 | 109.00 | 743.24 | 109.00 | 486.06 |
| eg_int_s | -3.41 | 6.45 | 7.46 | 80.71 | 6.45 | 117.52 | 6.45 | 105.28 | 6.45 | 111.39 | 6.45 | 131.01 | 6.45 | 139.73 | 100000.00 | 1800 | 8.09 | 1800 |
| ex14_1_6 | 0.00 | 0.00 | 1.00 | 0.01 | 0.00 | 0.24 | 0.00 | 0.19 | 0.00 | 0.15 | 0.00 | 0.21 | 0.00 | 0.17 | 0.00 | 0.06 | 0.00 | 0.01 |
| ex5_2_2_case1 | -400.00 | -400.00 | 0.00 | 0.02 | -400.00 | 0.19 | -400.00 | 0.15 | -400.00 | 0.14 | -400.00 | 0.13 | -400.00 | 0.15 | -400.00 | 0.05 | -400.00 | 0.13 |
| ex5_2_2_case2 | -600.00 | -600.00 | 0.00 | 0.02 | -600.00 | 0.10 | -600.00 | 0.11 | -600.00 | 0.11 | -600.00 | 0.10 | -600.00 | 0.11 | -600.00 | 0.17 | -600.00 | 0.12 |
| ex5_2_2_case3 | -750.00 | -750.00 | 0.00 | 0.02 | -750.00 | 0.09 | -750.00 | 0.10 | -750.00 | 0.11 | -750.00 | 0.11 | -750.00 | 0.12 | -750.00 | 0.04 | -750.00 | 0.05 |
| ex5_3_3 | 2.23 | 3.23 | 6.51 | 0.36 | inf | 0.65 | inf | 0.79 | inf | 0.98 | inf | 0.71 | inf | 1.14 | – | 1800 | 3.29 | 1800 |
| ex5_4_3 | 4845.46 | 4845.46 | 5937.44 | 0.02 | 4845.46 | 0.14 | 4845.46 | 0.10 | 4845.46 | 0.13 | 4845.46 | 0.12 | 4845.46 | 0.19 | 4845.46 | 0.01 | 4845.46 | 0.17 |
| ex5_4_4 | 10077.78 | 10077.78 | 11841.61 | 0.04 | 10077.78 | 0.26 | 10077.78 | 0.28 | 10077.78 | 0.2 | 10077.78 | 0.23 | 10077.78 | 0.18 | 10077.78 | 20.55 | 10077.73 | 6.06 |
| ex7_3_4 | 6.27 | 6.27 | 10.00 | 0.03 | 8.46 | 0.70 | inf | 0.74 | 8.46 | 1.12 | 8.46 | 0.61 | 8.46 | 0.85 | 6.27 | 1800 | 6.27 | 1.55 |
| ex8_3_13 | -100.00 | -43.09 | 0.00 | 0.34 | inf | 2.23 | -42.65 | 2.75 | -41.92 | 3.11 | inf | 2.15 | inf | 1.96 | – | 1800 | – | – |
| genpooling_lee2 | -3849.27 | -3849.27 | -3818.02 | 2.01 | -3516.59 | 76.09 | -3849.27 | 102.43 | -3849.27 | 97.85 | -3838.67 | 94.65 | -3604.08 | 122.55 | -3849.27 | 1800 | -3849.27 | 57.02 |
| graphpart_clique-70 | 1987.00 | 6348.00 | 6530.00 | 1814.66 | 8925.00 | 1189.03 | 8925.00 | 1294.08 | 8925.00 | 1270.16 | 8925.00 | 1313.94 | 8925.00 | 1200.14 | 9068.00 | 1800 | 8899.00 | 1800 |
| kall_circles_c7a | 2.66 | 2.66 | 4.11 | 0.4 | inf | 0.96 | inf | 0.80 | inf | 1.2 | inf | 0.70 | 3.56 | 1.53 | – | 1800 | 2.66 | 1.75 |
| mathopt5_6 | — | -0.94 | 2.56 | 0 | -0.63 | 0.04 | -0.94 | 0.06 | -0.94 | 0.05 | -0.94 | 0.06 | -0.94 | 0.05 | – | – | -0.94 | 0.01 |
| mhw4d | — | 0.03 | 27.87 | 0.01 | 0.03 | 0.16 | 0.03 | 0.21 | 0.03 | 0.15 | 0.03 | 0.16 | 0.03 | 0.14 | – | – | 0.03 | 0.14 |
| sfacloc1_2_90 | 5 | 17.89 | 24.66 | 0.99 | 17.89 | 5.25 | 17.89 | 5.50 | 17.89 | 4.86 | 17.89 | 11.18 | 17.89 | 5.81 | 22.33 | 1800 | 18.17 | 1800 |
| sfacloc1_4_90 | 0 | 10.46 | 16.72 | 4.38 | 10.55 | 14.71 | 10.67 | 13.53 | 10.61 | 13.93 | 10.54 | 25.26 | 10.46 | 22.49 | 22.40 | 1800 | 20.44 | 1800 |
| shiporig | 0 | 5.54 | 5.54 | 0.02 | inf | 0.28 | inf | 0.13 | inf | 0.19 | inf | 0.24 | inf | 0.06 | – | 1800 | – | – |
| sssd20-08persp | 279644 | 469667.30 | 471898.84 | 1800.04 | inf | 1.26 | 536480.83 | 256.87 | inf | 1.83 | 473060.97 | 288.57 | 484459.61 | 319.39 | 495881.90 | 1800 | 662341.70 | 1800 |
| sssd25-04persp | 287232 | 300186.80 | 300392.77 | 1800.03 | inf | 1.33 | 300768.37 | 252.13 | inf | 2.33 | 300768.37 | 247.79 | 301037.93 | 249.59 | 301790.80 | 1800 | 302000.93 | 1800 |
| st_bpv2 | -8 | -8.00 | 0.00 | 0.01 | -8.00 | 0.06 | -8.00 | 0.07 | -8.00 | 0.06 | -8.00 | 0.06 | -8.00 | 0.07 | -8.00 | 0 | -8.00 | 0 |
| st_bsj4 | -70262 | -70262.05 | -67897.66 | 0.02 | -70262.05 | 0.16 | -70262.05 | 0.13 | -70262.05 | 0.15 | -70262.05 | 0.15 | -70262.05 | 0.09 | -70262.05 | 0 | -70262.05 | 0.02 |

**Table 4.3:** Performance of schemes and solvers on MINLPLib instances (III)

| Name | Dual Bd | Primal Bd | bnb | | msbnbS1 | | msbnbS2 | | msbnbS3 | | msbnbS4 | | msbnbS5 | | scip | | couenne | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | $z^*$ | time | $z^*$ | time | $z^*$ | time | $z^*$ | time | $z^*$ | time | $z^*$ | time | $z^*$ | time | $z^*$ | time |
| st_e16 | 12292 | 12292.47 | 13680.79 | 0.02 | 12292.47 | 0.11 | 12292.47 | 0.10 | 12292.47 | 0.11 | 12292.47 | 0.11 | 12292.47 | 0.21 | 12292.47 | 0.06 | 12292.47 | 0.12 |
| st_e19 | -119 | -118.70 | -86.42 | 0.01 | -118.70 | 0.08 | -118.70 | 0.10 | -118.70 | 0.1 | -118.70 | 0.06 | -118.70 | 0.06 | -118.70 | 5.12 | -118.71 | 0.05 |
| st_fp7b | -635 | -634.75 | -490.11 | 0.06 | -634.75 | 0.18 | -634.75 | 0.16 | -634.75 | 0.2 | -634.75 | 0.20 | -634.75 | 0.23 | -634.75 | 0.08 | -634.75 | 0.29 |
| st_fp7c | -8695 | -8695.01 | -5136.40 | 0.05 | -8695.01 | 0.20 | -7251.69 | 0.20 | -8695.01 | 0.18 | -8695.01 | 0.22 | -8695.01 | 0.28 | -8695.01 | 0.06 | -8695.01 | 0.23 |
| st_fp7d | -115 | -114.75 | 6.55 | 0.02 | -114.75 | 0.16 | -114.75 | 0.19 | -114.75 | 0.19 | -114.75 | 0.19 | -114.75 | 0.22 | -114.75 | 0.06 | -114.75 | 0.3 |
| st_fp8 | 15639 | 15639.00 | 18423.00 | 0.02 | inf | 0.24 | inf | 0.32 | inf | 0.26 | inf | 0.25 | 15639.00 | 0.23 | 15639.00 | 0 | 15639.00 | 0.23 |
| st_glmp_fp3 | -12 | -12.00 | -9.00 | 0.01 | -12.00 | 0.06 | -12.00 | 0.06 | -12.00 | 0.07 | -12.00 | 0.08 | -12.00 | 0.08 | -12.00 | 0 | -12.00 | 0 |
| st_glmp_kk92 | -12 | -12.00 | -9.00 | 0.02 | -12.00 | 0.08 | -12.00 | 0.08 | -12.00 | 0.07 | -12.00 | 0.06 | -12.00 | 0.09 | -12.00 | 0 | -12.00 | 0.01 |
| st_ht | -2 | -1.60 | 0.00 | 0.01 | -1.45 | 0.05 | -1.60 | 0.06 | -1.60 | 0.05 | -1.60 | 0.05 | -1.60 | 0.05 | -1.60 | 0 | -1.60 | 0.02 |
| st_jcbpaf2 | -795 | -794.86 | -388.35 | 0.02 | -794.86 | 0.13 | -794.86 | 0.13 | -794.86 | 0.11 | -794.86 | 0.12 | -388.35 | 0.1 | -794.86 | 0.01 | -794.86 | 0.04 |
| st_pan1 | -5 | -5.28 | 0.00 | 0.01 | -4.70 | 0.07 | -4.62 | 0.06 | -5.28 | 0.09 | -5.28 | 0.08 | -5.28 | 0.07 | -5.28 | 0 | -5.28 | 0.02 |
| st_ph11 | -11 | -11.28 | 0.00 | 0.01 | -11.28 | 0.06 | -11.28 | 0.06 | -11.28 | 0.06 | -8.00 | 0.07 | -11.28 | 0.06 | -11.28 | 0 | -11.28 | 0.02 |
| st_ph12 | -23 | -22.63 | 0.00 | 0.01 | -22.63 | 0.06 | -22.63 | 0.11 | -22.63 | 0.09 | -22.63 | 0.09 | -22.63 | 0.06 | -22.63 | 0 | -22.63 | 0.01 |
| st_ph13 | -11 | -11.28 | 0.00 | 0.01 | -11.28 | 0.08 | -11.28 | 0.08 | -11.28 | 0.09 | -11.28 | 0.07 | -11.28 | 0.06 | -11.28 | 0 | -11.28 | 0.01 |
| st_ph15 | -393 | -392.70 | 0.00 | 0.01 | -392.70 | 0.08 | -392.70 | 0.09 | -392.70 | 0.09 | -392.70 | 0.08 | -392.70 | 0.08 | -392.70 | 0 | -392.70 | 0.01 |
| st_ph1 | -230 | -230.12 | 0.00 | 0.01 | -230.12 | 0.11 | -230.12 | 0.11 | -230.12 | 0.11 | -230.12 | 0.09 | -215.52 | 0.14 | -230.12 | 0 | -230.12 | 0.02 |
| st_ph2 | -1028 | -1028.12 | 0.00 | 0.01 | -932.86 | 0.15 | -1028.12 | 0.13 | -1028.12 | 0.14 | -1028.12 | 0.12 | -1028.12 | 0.13 | -1028.12 | 0 | -1028.12 | 0.02 |
| st_ph3 | -420 | -420.23 | 0.00 | 0.01 | -420.23 | 0.07 | -420.23 | 0.09 | -420.23 | 0.11 | -420.23 | 0.09 | -420.23 | 0.1 | -420.23 | 0 | -420.23 | 0.01 |
| st_qpc-m0 | -5 | -5.00 | 0.00 | 0 | -5.00 | 0.05 | -5.00 | 0.05 | -5.00 | 0.06 | -5.00 | 0.05 | -5.00 | 0.07 | -5.00 | 0 | -5.00 | 0.01 |
| st_qpk1 | -3 | -3.00 | 0.00 | 0.01 | -3.00 | 0.06 | -3.00 | 0.06 | -3.00 | 0.07 | -3.00 | 0.06 | -3.00 | 0.07 | -3.00 | 0 | -3.00 | 0.02 |
| st_rv2 | -64 | -64.48 | -62.75 | 0.01 | -64.48 | 0.13 | -64.48 | 0.13 | -64.48 | 0.14 | -64.48 | 0.13 | -64.48 | 0.16 | -64.48 | 0.01 | -64.48 | 0.32 |
| st_rv9 | -120 | -120.15 | -117.59 | 0.21 | -118.98 | 0.28 | -118.98 | 0.35 | -118.98 | 0.32 | -118.98 | 0.26 | -117.59 | 2.9 | -120.15 | 0.14 | -120.15 | 3.24 |
| supplychainp1_020306 | 437552 | 437551.68 | 438471.69 | 1.12 | 437551.63 | 7.00 | 437551.63 | 5.96 | 437551.63 | 6.66 | 437551.63 | 6.89 | 437551.63 | 7.12 | 437551.68 | 0.2 | 437551.68 | 0.95 |
| tricp | 0 | 0.00 | 263478.49 | 3.67 | 61.25 | 11.85 | 0.00 | 9.88 | 740.89 | 7.72 | 0.00 | 9.43 | 0.00 | 7.37 | 88750000.00 | 1800 | 740.89 | 1800 |
| trig | — | -3.76 | 0.00 | 0.01 | -2.48 | 0.06 | -3.76 | 0.06 | -3.76 | 0.06 | -3.76 | 0.06 | -3.56 | 0.06 | – | – | -3.76 | 0.03 |
| tspn15 | 299 | 327.14 | 329.91 | 23.83 | 327.14 | 18.73 | 327.14 | 23.00 | 327.14 | 20.99 | 327.14 | 18.72 | 327.14 | 27.29 | – | 1800 | 331.09 | 1800 |
| wall | -1 | -1.00 | 1.00 | 0.01 | 0.00 | 0.13 | 1.00 | 0.37 | inf | 0.25 | inf | 0.37 | 1.00 | 0.12 | – | – | – | 1.96 |
| wastewater05m1 | 230 | 229.70 | 269.43 | 0.19 | 229.70 | 0.60 | 260.37 | 1.00 | 229.70 | 1.15 | 542.20 | 0.75 | 260.37 | 1.18 | 229.70 | 123.69 | 229.70 | 400.75 |
| wastewater11m1 | 1516 | 2127.12 | 3080.27 | 0.27 | 2127.12 | 0.70 | 2127.12 | 0.75 | 2127.12 | 0.58 | 2127.12 | 0.52 | 2127.12 | 0.72 | 4405.16 | 1800 | 2784.97 | 1800 |
| wastewater13m1 | 1244 | 1564.96 | 1842.33 | 0.97 | 1564.96 | 1.03 | 1564.96 | 1.16 | 1564.96 | 1.17 | 1564.96 | 1.40 | 1564.96 | 1.17 | 2031.93 | 1800 | 1564.96 | 1800 |
| wastewater14m1 | 383 | 513.00 | 567.20 | 0.14 | 513.00 | 0.38 | 513.00 | 0.56 | 513.00 | 0.41 | 513.00 | 0.62 | 513.00 | 0.46 | 578.90 | 1800 | 536.53 | 1800 |
| blend146 | 47 | 45.28 | – | – | -inf | 352.36 | -inf | 347.89 | 40.54 | 360.69 | -inf | 348.44 | -inf | 292.16 | 43.82 | 1800 | 45.28 | 1800 |
| blend852 | 54 | 53.96 | – | – | 51.19 | 390.49 | 51.07 | 376.92 | -inf | 388.21 | -inf | 400.70 | 47.66 | 371.05 | 53.30 | 1800 | 53.96 | 1433.22 |
| sportfournament14 | 96 | 96.00 | 90.00 | 0.14 | 94.00 | 2.64 | 94.00 | 0.81 | 94.00 | 1.94 | 94.00 | 3.98 | 94.00 | 1.59 | 94.00 | 1800 | 96.00 | 233.56 |

# Chapter 5

# Mixed-Integer Derivative-Free Optimization

We study the problem of minimizing a convex function on a finite subset of the integer lattice (**?**). In particular, we consider problems of the form

$$\underset{x}{\text{minimize}} \;\; f(x) \quad \text{subject to } x \in \Omega \subset \mathbb{Z}^n. \tag{5.1}$$

We first define what it means for $f$ to be convex on a set $\Omega \subset \mathbb{Z}^n$.

**Definition 5.0.1** (Convexity on Integer Subsets)**.** *A function $f$ is* (integer-)convex on $\Omega$ *if for $x \in \Omega$ and any $p$ points $y^i \in \Omega$ satisfying $x = \sum_{i=1}^{p} \lambda_i y^i$ with $\lambda_i \in [0, 1]$ and $\sum_{i=1}^{p} \lambda_i = 1$, then $f(x) \leq \sum_{i=1}^{p} \lambda_i f(y^i)$.*

Note that if no point in $\Omega$ can be represented as a convex combination of other points (e.g., if $\Omega$ represents purely binary decisions), then any function trivially satisfies Definition 5.0.1.

We make the following assumption about Problem (5.1).

**Assumption 5.1.** *$f$ is convex on $\Omega$, $\Omega \subset \mathbb{Z}^n$ is nonempty and bounded, and $f$ cannot be evaluated at $x \notin \Omega$.*

Because we assume that $f$ cannot be evaluated at noninteger points, Problem (5.1) can be referred to as a convex optimization problem with *unrelaxable integer constraints* (**?**). We note that $\Omega$ needs not contain all integer points in its convex hull (i.e., our approach allows for situations where $\text{conv}(\Omega) \cap \mathbb{Z}^n \neq \Omega$); such instances may arise when arguments to $f$ must satisfy some additional constraints. Problem (5.1) can also be viewed as minimizing an *integer-convex*[1] function ((**?**, Definition 15.2)) over a nonempty finite subset of $\mathbb{Z}^n$.

[1] Although **?** considers integer convexity only for polynomials, this definition can be applied to more general classes of functions over the sets considered here.

Admittedly, it is rare to know that $f$ is convex when $f$ is not given in closed form (although one may be able to detect convexity (**?**) or estimate a probability that $f$ is convex on the finite domain $\Omega$ (**?**). Nevertheless, studying the convex case is important because we are unaware of any method (besides complete enumeration) for obtaining exact solutions to (5.1) when $f$ (convex or otherwise) cannot be evaluated at noninteger points.

One example where an objective is not given in closed form but is known to be convex arises in the combinatorial optimal control of partial differential equations (PDEs). For example, (**?**, Lemma 2) show that the solution operator of certain semilinear elliptic PDEs is a convex function of the controls provided that the nonlinearities in the PDE and boundary conditions are concave and nondecreasing. Thus, any linear function of the states of the PDE (e.g., the max-function) is a convex function of the controls when the states are eliminated. **?** propose using adjoint information to compute subgradients of the (continuous relaxation of the) objective, but an alternative would be to consider a derivative-free approach.

We consider only pure-integer problems of the form (5.1); however, our developments are equally applicable to the mixed-integer case

$$\underset{x,y}{\text{minimize}} \;\; F(x,y) \quad \text{subject to } (x,y) \in \Omega \times \Psi \subset \mathbb{Z}^n \times \mathbb{R}^m \tag{5.2}$$

provided $F$ is convex on $\Omega \times \Psi$. If we define the function

$$f(x) = \min_{y \in \Psi} F(x,y)$$

and if $f$ is well defined over $\Omega$, then (5.2) can be solved by minimizing $f$ on $\Omega \subset \mathbb{Z}^n$, where each evaluation $f(x)$ requires an optimization of the continuous variables $y$ for a fixed $x$. (When there is no additional information on $F$ and $\Psi$, each continuous optimization problem may be difficult to solve.) Because many of the results below rely only on the convexity of $f$ and not the discrete nature of $\Omega$, much of the analysis below readily applies to the mixed-integer case.

We are especially interested in problems where the cost of evaluating $f$ is large. Problems of the form (5.1) or (5.2) where the objective is expensive to evaluate and some integer constraints are unrelaxable arise in a range of simulation-based optimization problems. For example, the optimal design of concentrating solar power plants gives rise to computationally expensive simulations for each set of design parameters (**?**). Furthermore, some of the design parameters (e.g., the number of panels on the power plant receiver) cannot be relaxed to noninteger values. Similar problems arise when tuning codes to run on high-performance computers (**?**). In this case, $f(x)$ may be the memory footprint of a

code that is compiled with settings $x$, which can correspond to decisions such as loop unrolling or tiling factors that do not have meaningful noninteger values. Optimal material design problems may also constrain the choice of atoms to a finite set, resulting in unrelaxable integer constraints; **?** propose a derivative-free optimization algorithm designed explicitly for such a problem.

Motivated by such applications, we develop a method that will certifiably converge to the solution of (5.1) under Assumption 5.1 without access to $\partial f$ (i.e., a (sub)gradient of a continuous relaxation of $f$). Using only evaluations of $f$, we construct *secants*, which are linear functions that interpolate $f$ at a set of $n + 1$ points. These secant functions underestimate $f$ in certain parts of $\Omega$. We use these secants to define *conditional cuts* that are valid in disconnected portions of the domain. The complete set of secants and the conditions that describe when they are valid are used to construct an underestimator of a convex $f$. While access to $\partial f$ could strengthen such an underestimator, we do not address such considerations in this paper. While different specific applications are by nature convex, in many other situations, one might suspect that $f$ is convex, but without proof. In such cases, if $f$ is indeed convex, our method will converge to a global optimum. Otherwise, our algorithm might generate cuts that are not valid, hence, the model might not be an underestimator of $f$. However, our algorithm will still converge to a (possibly good) local minimum, but will not be able to provide a proof of (convexity or) nonconvexity of $f$. Also, our method can be applied in a specific restricted region of a known nonconvex problem, in which case it will converge to a local optimum in that region.

While we consider only bound constraints solving (5.1) in this paper, our method can easily be extended to cater to general constraints.

Solving (5.1) under Assumption 5.1 without access to $\partial f$ poses a number of theoretical and computational challenges. Because the integer constraints are unrelaxable, one cannot apply traditional branch-and-bound approaches. In particular, model-based continuous derivative-free methods would require evaluating the objective at noninteger points to ensure convergence for the continuous relaxation of (5.1). In addition, other traditional techniques for mixed-integer optimization—such as Benders decomposition (**?**) or outer approximation by **?** and **?**—cannot be used to solve (5.1) when $\partial f$ is unavailable. Since we know of no method (other than complete enumeration) for obtaining global minimizers of (5.1) under Assumption 5.1, we know of no potential algorithm to address this problem when a (sub)gradient is unavailable.

We make three contributions in this paper: (1) we develop a new underestimator for convex functions on subsets of the integer lattice that is based solely on function evaluations; (2) we present an algorithm that alternates between updating this underestimator and evaluating the objective in order to identify a global solution of Problem (5.1) under Assumption 5.1; and (3) we show empirically that certifying global optimality in such cases is a challenging problem. In our experiments, we are unable to prove optimality for many problems when $n \geq 5$, and we provide insights into why a proof of optimality remains computationally challenging.

**Outline.** Section 5.1 surveys recent methods for addressing (5.1). Section 5.2 introduces valid conditional cuts using only the function values of a convex objective and discusses the theoretical properties of these cuts. Section 5.3 presents an algorithm for solving (5.1) and shows that this algorithm identifies a global minimizer of (5.1) under Assumption 5.1. Section 5.4 considers an MILP based approach for formulating the underestimator and Section 5.5 presents the method *SUCIL*—secant underestimator of convex functions on the integer lattice. Section 5.6 provides detailed numerical studies for implementations of *SUCIL* on a set of convex problems. Section 5.7 discusses many of the challenges in obtaining global solutions to (5.1).

## 5.1 Background

Developing methods to solve (5.1) without access to derivatives of $f$ is an active area of research (**???**) Most methods address general (i.e., nonconvex) functions $f$, and heuristic approaches are commonly adopted to handle integer decision variables for such derivative-free optimization problems. For example, the method by **?** rounds noninteger components of candidate points to the nearest feasible integer values. The method's asymptotic convergence results are based on the inclusion of points drawn uniformly from the finite domain (and rounding noninteger values as necessary).

Integer-constrained pattern-search methods by **?** and **?** generalize their continuous counterparts. These modified pattern-search methods can be shown to converge to *mesh-isolated minimizers*: points with function values that are better than all neighboring points on the integer lattice. Unfortunately, such mesh-isolated minimizers can be arbitrarily far from a global minimizer, even when $f$

is convex; **?** (Figure 2) show an example of such a function $f$. Other methods that converge to mesh-isolated minimizers include direct-search methods that update the integer variables via a local search (**??**) and mesh adaptive direct-search methods adapted to address discrete and granular variables (i.e., those that have a controlled number of decimals) (**??**). The direct-search method by **?** accounts for integer constraints by constructing a set of directions that have a nonnegative span of $\mathbb{R}^n$ and that ensure that all evaluated points will be integer valued. This method is shown to converge to a type of stationary point that, even in the convex case, may not be a global minimizer. **?** present various definitions of local minimizers of (5.1) and a discussion of associated properties. The BFO method by **?** has a recursive step that explores points near the current iterate by fixing each of the discrete variables to its value plus or minus a step-size parameter.

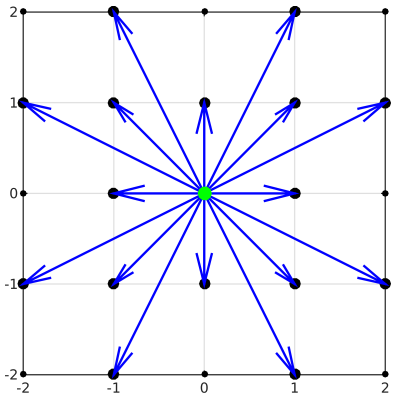



**Figure 5.1:** 16 primitive directions emanating from $(0, 0)$ in the domain $\Omega = [-2, 2]^2 \cap \mathbb{Z}^2$

| | $n = 2$ | | $n = 3$ | | $n = 4$ | | $n = 5$ | |
|---|---|---|---|---|---|---|---|---|
| $k$ | $\lvert\Omega\rvert$ | # | $\lvert\Omega\rvert$ | # | $\lvert\Omega\rvert$ | # | $\lvert\Omega\rvert$ | # |
| 1 | 9 | 8 | 27 | 26 | 81 | 80 | 243 | 242 |
| 2 | 25 | 16 | 125 | 98 | 625 | 544 | 3,125 | 2,882 |
| 3 | 49 | 32 | 343 | 290 | 2,403 | 2,240 | 16,807 | 16,322 |
| 4 | 81 | 48 | 729 | 578 | 6,561 | 5,856 | 59,049 | 55,682 |

**Table 5.1:** Number of primitive directions in a discrete 1-neighborhood, # = $|\mathcal{N}(x_c, 1)|$, that emanate from the origin $x_c$ of the domain $\Omega = [-k, k]^n \cap \mathbb{Z}^n$ and that correspond to points in $\Omega$.

The method by **?** uses line searches over a set of *primitive directions*, that is, a set of scaled directions $D$ where no vector $d_i \in D$ is a positive multiple of a different $d_j \in D$. This method explores a discrete set of directions around the current iterate until finding a local minimum $x_c$ in a $\beta$-neighborhood, defined as $\mathcal{N}(x_c, \beta) = \{x_c + \alpha d \in \Omega : d \in D, \alpha \in \mathbb{N}, \alpha \leq \beta\}$ for $\beta \in \mathbb{N}$. Although **?** target nonconvex objectives, their approach will converge to a global minimum $x_c$ of a convex objective $f$ if all points in $\mathcal{N}(x_c, 1)$ are evaluated. Figure 5.1 illustrates such a discrete 1-neighborhood. Unfortunately, $|\mathcal{N}(x_c, 1)|$ can be large; see Table 5.1.

Model-based methods approximate objective functions on the integer lattice by using surrogate models (see, for example, **?**, **?**, and **?**). The surrogate model is used to determine points where the objective should be evaluated; the model is typically refined after each objective evaluation. The methodology by **?** specifi-

cally uses radial basis function surrogate models and does automatic model selection at each iteration. Mixed-integer nonlinear optimization solvers are used to minimize the surrogate to obtain the next integer point for evaluation. The model-based methods of **??** and **?** modify the sampling strategies and local searches typically used to solve continuous objective versions. The approaches by **?** and **?** restart when a suitably defined local minimizer is encountered, continuing to evaluate the objective until the available budget of function evaluations is exhausted. These model-based methods differ in the initial sampling method, the type of surrogate model, and the sampling strategy used to select the next points to be evaluated. **?** present a survey and taxonomy of continuous and integer model-based optimization approaches.

In a different line of research, **?** propose a branch-and-bound framework to address binary variables; a solution to the relaxed nonlinear subproblems is obtained via a combination of global kriging models and local surrogate models. Similarly, **?** replace the black-box portions of the objective function (and constraints) by a stochastic surrogate; the resulting mixed-integer nonlinear programs are solved by branch and bound. Both approaches assume that the integer constraints are relaxable.

## 5.2 Underestimator of Convex Functions on the Integer Lattice

To construct an underestimator of a convex objective function $f$, we now discuss secant functions, which are linear mappings that interpolate $f$ at $n+1$ points. We provide conditions for where these cuts will underestimate $f$. We then discuss necessary conditions on the set of evaluated points so that if all possible secants are constructed, these conditional cuts underestimate $f$ on the domain $\Omega$. As we will see in Section 5.3, this underestimator is essential for obtaining a global minimizer of (5.1) under Assumption 5.1. Throughout this section, $X \subseteq \Omega$ denotes a set of at least $n+1$ points in $\Omega$ at which the objective function $f$ has been evaluated.

### 5.2.1 Secant Functions and Conditional Cuts

Constructing a secant function requires a set of $n+1$ interpolation points where $f$ has been evaluated. To define a secant function for $f$, we introduce a multi-index $\boldsymbol{i}$ of $n+1$ distinct indices, $1 \leq i_1 < \ldots < i_{n+1} \leq |X|$, as $\boldsymbol{i} = (i_1, \ldots, i_{n+1})$. With a slight abuse of notation, we will refer to elements $i_j \in \boldsymbol{i}$.

Given the set of points $X^{\boldsymbol{i}} = \left\{x^{i_j} \in X \subseteq \Omega : i_j \in \boldsymbol{i}\right\}$, we construct the secant function

$$m^{\boldsymbol{i}}(x) = (c^{\boldsymbol{i}})^\top x + b^{\boldsymbol{i}},$$

where the coefficients $c^{\boldsymbol{i}} \in \mathbb{R}^n$ and $b^{\boldsymbol{i}} \in \mathbb{R}$ are the solution to the linear system

$$\left[\bar{X}^{\boldsymbol{i}}\ e\right]\begin{bmatrix} c^{\boldsymbol{i}} \\ b^{\boldsymbol{i}} \end{bmatrix} = f^{\boldsymbol{i}}, \text{ where } \bar{X}^{\boldsymbol{i}} = \begin{bmatrix} (x^{i_1})^\top \\ \vdots \\ (x^{i_{n+1}})^\top \end{bmatrix}, \ e = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}, \text{ and } f^{\boldsymbol{i}} = \begin{bmatrix} f(x^{i_1}) \\ \vdots \\ f(x^{i_{n+1}}) \end{bmatrix}. \quad (5.3)$$

The secant function $m^{\boldsymbol{i}}$ is unique provided that the set $X^{\boldsymbol{i}}$ is *affinely independent*. We now show that the secant function $m^{\boldsymbol{i}}$ underestimates $f$ in certain polyhedral cones, namely, the cones

$$\mathcal{U}^{\boldsymbol{i}} = \bigcup_{i_j \in \boldsymbol{i}} \operatorname{cone}(x^{i_j} - X^{\boldsymbol{i}}), \quad (5.4)$$

where

$$\operatorname{cone}(x^{i_j} - X^{\boldsymbol{i}}) = \left\{ x^{i_j} + \sum_{l=1, l\neq j}^{n+1} \lambda_l (x^{i_j} - x^{i_l}) : \ i_j \in \boldsymbol{i}, i_l \in \boldsymbol{i}, \lambda_l \geq 0 \right\}. \quad (5.5)$$

**Lemma 5.2.1** (Conditional cuts)**.** *If $f$ is convex on $\Omega$ and $X^{\boldsymbol{i}} \subseteq \Omega$ is affinely independent, then the unique linear mapping $m^{\boldsymbol{i}}$ satisfying $m^{\boldsymbol{i}}(x^{i_j}) = f(x^{i_j})$ for each $i_j \in \boldsymbol{i}$ satisfies $m^{\boldsymbol{i}}(x) \leq f(x)$ for all $x \in \mathcal{U}^{\boldsymbol{i}} \bigcap \Omega$.*

*Proof.* The uniqueness of the linear mapping follows directly from the affine independence of $X^{\boldsymbol{i}}$.

Let $x$ be a point in $\operatorname{cone}(x^{i_j} - X^{\boldsymbol{i}}) \bigcap \Omega$ for arbitrary $x^{i_j} \in X^{\boldsymbol{i}}$. By (5.5),

$$x = x^{i_j} + \sum_{l=1, l\neq j}^{n+1} \lambda_l \left(x^{i_j} - x^{i_l}\right), \quad (5.6)$$

with $\lambda_l \geq 0$ (for $l = 1, \ldots, n+1; l \neq j$). Rearranging (5.6) yields

$$x^{i_j} = \frac{1}{1 + \sum_{k=1,k\neq j}^{n+1} \lambda_k} x + \frac{1}{1 + \sum_{k=1,k\neq j}^{n+1} \lambda_k} \sum_{l=1,l\neq j}^{n+1} \lambda_l x^{i_l},$$

showing that $x^{i_j}$ can be expressed as a convex combination of $\{x\} \cup \{x^{i_l} : l = 1, \ldots, n+1; l \neq j\}$, all of which are points in $\Omega$. Therefore, by convexity of $f$ on $\Omega$ (see Definition 5.0.1),

$$f(x^{i_j}) \leq \frac{1}{1 + \sum_{k=1,k\neq j}^{n+1} \lambda_k} f(x) + \frac{1}{1 + \sum_{k=1,k\neq j}^{n+1} \lambda_k} \sum_{l=1,l\neq j}^{n+1} \lambda_l f\left(x^{i_l}\right).$$

Solving for $f(x)$ and using the fact that $m^{\boldsymbol{i}}$ interpolates $f$ at points in $X^{\boldsymbol{i}}$, we obtain

$$\begin{aligned} f(x) &\geq \left(1 + \sum_{k=1,k\neq j}^{n+1} \lambda_k\right) f(x^{i_j}) - \sum_{l=1,l\neq j}^{n+1} \lambda_l f(x^{i_l}), \\ &= \left(1 + \sum_{k=1,k\neq j}^{n+1} \lambda_k\right) m^{\boldsymbol{i}}(x^{i_j}) - \sum_{l=1,l\neq j}^{n+1} \lambda_l m^{\boldsymbol{i}}(x^{i_l}), \\ &= m^{\boldsymbol{i}}(x^{i_j}) + \sum_{l=1,l\neq j}^{n+1} \lambda_l \left(m^{\boldsymbol{i}}(x^{i_j}) - m^{\boldsymbol{i}}(x^{i_l})\right), \\ &= m^{\boldsymbol{i}}(x), \end{aligned}$$

where the last equality holds by (5.6) and the linearity of $m^{\boldsymbol{i}}$. Because $x$ is an arbitrary point in $\text{cone}(x^{i_j} - X^{\boldsymbol{i}})$ for arbitrary $x^{i_j}$, the result is shown. □

**Corollary 5.2.2.** *The linear mapping $m^{\boldsymbol{i}}(x)$ defined in Lemma 5.2.1 satisfies $f(x) \geq m^{\boldsymbol{i}}(x),\ \forall x \in \text{cone}(x^k - X^{\boldsymbol{i}} \setminus x^{i_l})$, where $x^k$ is such that $f(x^k) \geq f(x^{i_l})$.*

Corollary 5.2.2 indicates that one can just replace a point in $X^{\boldsymbol{i}}$ and the earlier mapping $m^{\boldsymbol{i}}$ would still be valid in the cone associated with the new point. However, the mapping $m^{\boldsymbol{i}}(x)$ would yield a 'weaker' underestimator than the one derived using the points $X^{\boldsymbol{i}} \cup x^{i_k} \setminus x^{i_l}$.

**Corollary 5.2.3.** *Let $x^k$ be a point such that the points $X^{\boldsymbol{i}'} = X^{\boldsymbol{i}} \cup x^k \setminus x^{i_l}$ are affinely independent and $f(x^k) \leq f(x^{i_j})$. Then, the linear mapping $m^{\boldsymbol{i}'}(x)$ derived using the points $X^{\boldsymbol{i}'}$ as per Lemma 5.2.1 satisfies $f(x) \geq m^{\boldsymbol{i}'}(x),\ \forall x \in \mathcal{U}^{\boldsymbol{i}} \setminus \text{cone}(x^k - X^{\boldsymbol{i}})$.*

Corollary 5.2.3 states that a valid (and tighter) cut can be obtained for the cones associated with the points $X^{\boldsymbol{i}} \setminus x^{i_l}$ by replacing one of the points $x^{i_l}$ in $X^{\boldsymbol{i}}$ by another point with a function value lower than that of $x^{i_l}$. The proofs of Corollary 5.2.2 and Corollary 5.2.3 follow easily from that of Lemma 5.2.1.

We now prove that the cones in $\mathcal{U}^{\boldsymbol{i}}$ do not intersect when $X^{\boldsymbol{i}}$ is affinely independent. We note that the following result holds for any affinely independent set $X^{\boldsymbol{i}} \subset \mathbb{R}^n$.

**Lemma 5.2.4** (A point is in at most one cone)**.** *If $X^{\boldsymbol{i}}$ is affinely independent, no point $x \in \mathbb{R}^n$ satisfies $x \in \text{cone}(x^{i_j} - X^{\boldsymbol{i}})$ and $x \in \text{cone}(x^{i_k} - X^{\boldsymbol{i}})$ for $x^{i_j}, x^{i_k} \in X^{\boldsymbol{i}}$ and $x^{i_j} \neq x^{i_k}$.*

*Proof.* Let $x^{i_1}$ and $x^{i_2}$ be different, but otherwise arbitrary, points in $X^{\boldsymbol{i}}$. To arrive at a contradiction, suppose that there exists $x \in \text{cone}(x^{i_1} - X^{\boldsymbol{i}}) \cap \text{cone}(x^{i_2} - X^{\boldsymbol{i}})$. That is, $x = x^{i_1} + \sum_{l=2}^{n+1} \lambda_l \left(x^{i_1} - x^{i_l}\right)$ and $x = x^{i_2} + \sum_{l=1,l\neq 2}^{n+1} \sigma_l \left(x^{i_2} - x^{i_l}\right)$ for $\lambda_l \geq 0$ $(l \in \{2, \ldots, n+1\})$

and $\sigma_l \geq 0$ ($l \in \{1, 3, \ldots, n+1\}$). Subtracting these two expressions yields

$$\begin{aligned}
0 &= x^{i_1} - x^{i_2} + \sum_{l=2}^{n+1} \lambda_l (x^{i_1} - x^{i_l}) - \sum_{l=1, l\neq 2}^{n+1} \sigma_l (x^{i_2} - x^{i_l}), \\
&= x^{i_1} - x^{i_2} + \sum_{l=2}^{n+1} \lambda_l x^{i_1} - \sum_{l=2}^{n+1} \lambda_l x^{i_2} + \sum_{l=2}^{n+1} \lambda_l x^{i_2} - \sum_{l=2}^{n+1} \lambda_l x^{i_l} - \sum_{l=1, l\neq 2}^{n+1} \sigma_l (x^{i_2} - x^{i_l}), \\
&= \left(1 + \sum_{l=2}^{n+1} \lambda_l\right)\left(x^{i_1} - x^{i_2}\right) - \sum_{l=2}^{n+1} \lambda_l \left(x^{i_l} - x^{i_2}\right) + \sum_{l=1, l\neq 2}^{n+1} \sigma_l \left(x^{i_l} - x^{i_2}\right), \\
&= \left(1 + \sigma_1 + \sum_{l=2}^{n+1} \lambda_l\right)\left(x^{i_1} - x^{i_2}\right) + \sum_{l=3}^{n+1} (\sigma_l - \lambda_l)\left(x^{i_l} - x^{i_2}\right). \qquad (5.7)
\end{aligned}$$

Since $X^{\boldsymbol{i}}$ is an affinely independent set, the vectors $\left\{x^{i_l} - x^{i_2} : i_l \in \boldsymbol{i}, i_l \neq i_2\right\}$ are linearly independent. Hence the dependence relation in (5.7) can be satisfied only if the coefficient on $(x^{i_1} - x^{i_2})$ vanishes. That is,

$$1 + \sigma_1 + \sum_{l=2}^{n+1} \lambda_l = 0,$$

which contradicts $\lambda_l \geq 0$ (for $l = 2, \ldots, n+1$) and $\sigma_1 \geq 0$. Since $x^{i_1}$ and $x^{i_2}$ were arbitrary points in $X^{\boldsymbol{i}}$, the result is shown. □

For each affinely independent set $X^{\boldsymbol{i}}$, Lemma 5.2.1 ensures that the secant function $m^{\boldsymbol{i}}$ underestimates $f$ on $\Omega \cap \mathcal{U}^{\boldsymbol{i}}$. We can therefore underestimate $f$ via a model that consists of the pointwise maximum of the underestimators for which the point is in $\mathcal{U}^{\boldsymbol{i}}$ for some affinely independent set $X^{\boldsymbol{i}}$ of previously evaluated points. Minimizing this nonconvex model on the integer $\Omega$ can then provide a lower bound for the global minimum of $f$ on $\Omega$.

Figure 5.2 shows a two-dimensional example of points that produce such secant functions and the regions in which they will underestimate any convex function $f$. For the $n+2$ points (circles), we consider three affinely independent sets indicated by triangles linking $n+1$ points. The left image shows an affinely independent set (blue line triangle), and the three cones (shaded blue area) in which the secant through these points is a valid underestimator. The right image shows that conditional cuts using $n+2$ points can cover all of $\mathbb{R}^n$.

### 5.2.2 Lower Bound for $f$

We now describe an optimization problem whose solution provides a lower bound for $f$ on $\Omega$. Let $W(X)$ denote the set of all multi-indices corresponding to

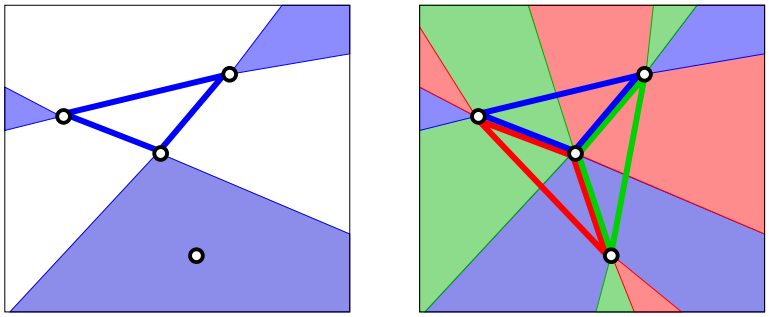


**Figure 5.2:** Illustration of areas in $\mathbb{R}^2$ where conditional cuts are valid. Left shows the regions of the domain where the secant through three points (the vertices of the blue triangle) will underestimate $f$. Right shows that one point in the interior of $n+1$ points is sufficient to underestimate $f$. The conditional cuts correspond to the $n+1$ points in the triangle of the same color.

affinely independent subsets of $X$:

$$W(X) = \left\{ \boldsymbol{i} : X^{\boldsymbol{i}} \subseteq X, X^{\boldsymbol{i}} \text{ is affinely independent} \right\}. \tag{5.8}$$

If $f$ has been evaluated at every point in $X$, we can construct a secant function $(c^{\boldsymbol{i}})^\top x + b^{\boldsymbol{i}}$ interpolating $f$ on $X^{\boldsymbol{i}}$ for every multi-index $\boldsymbol{i} \in W(X)$. We then collect all such conditional cuts in the piecewise integer linear program

$$\begin{aligned} &\underset{x,\eta}{\text{minimize}}\ \eta \\ &\text{subject to } \eta \geq (c^{\boldsymbol{i}})^\top x + b^{\boldsymbol{i}}, \text{ for all } \boldsymbol{i} \in W(X) \text{ with } x \in \mathcal{U}^{\boldsymbol{i}}, \\ &\qquad\qquad x \in \Omega, \end{aligned} \tag{PILP}$$

where $\mathcal{U}^{\boldsymbol{i}}$ is defined in (5.4).

For the set of points $X$ and corresponding $W(X)$, let $\eta(\bar{x})$ denote the value of (PILP) when the constraint $x = \bar{x}$ is added to (PILP) for a particular $\bar{x} \in \Omega$.

As we will see below, $\eta$ represents the largest lower bound for $f$ induced by the set $X$, and the solution to (PILP) provides a lower bound for $f$ on $\Omega$. Because the cuts in (PILP) are valid only within $\mathcal{U}^{\boldsymbol{i}}$, the resulting model takes an optimal value of $\eta_* = -\infty$ if there is a point $x \in \Omega$ that is not in the union of $\mathcal{U}^{\boldsymbol{i}}$ over all $\boldsymbol{i} \in W(X)$. Also, if size of $W(X)$ is sufficiently large, there might be a large number of redundant inequalities in (PILP), however, finding and removing such inequalities seems to be difficult and beyond the scope of this paper.

**Lemma 5.2.5** (Underestimator of $f$)**.** *If $f$ is convex on $\Omega$, then the optimal value $\eta_*$ of (PILP) satisfies $\eta_* \leq f(x)$ for all $x \in \Omega$.*

*Proof.* If $W(X)$ is empty, the result holds trivially since $\eta$ is unconstrained. Otherwise, since (PILP) minimizes $\eta$, it suffices to show that $\eta(x) \leq f(x)$ for arbitrary $x \in \Omega$. Two cases can occur. First, if $x \notin \mathcal{U}^{\boldsymbol{i}}$ for every $\boldsymbol{i} \in W(X)$, then no conditional cut exists at $x$. Thus $\eta(x) = -\infty$ and $\eta(x) < f(x)$. Second, if $x \in \mathcal{U}^{\boldsymbol{i}}$ for some $\boldsymbol{i} \in W(X)$, then

$$(c^{\boldsymbol{i}})^\top x + b^{\boldsymbol{i}} \leq f(x),$$

by Lemma 5.2.1, where the affine independence of $X^{\boldsymbol{i}}$ follows from the definition of $W(X)$. Therefore, $\eta(x) \leq f(x)$ for all $x \in \Omega$. Since $\eta_* = \min_{x \in \Omega} \eta(x)$, the result is shown. □

If $W(X)$ in (PILP) is replaced by a proper subset $W'(X) \subset W(X)$ of multi-indices, then Lemma 5.2.5 still holds. (This relaxation of (PILP) associated with removing constraints cannot increase $\eta_*$.) Such a replacement may be necessary if $W(X)$ becomes too large to allow considering every affinely independent subset of $X$ when forming (PILP).

### 5.2.3 Covering $\mathbb{R}^n$ with Conditional Cuts

We find it beneficial to ensure that $X$ contains points that result in a finite objective value for the underestimator described by (PILP). We now establish a condition that ensures that the union of conditional cuts induced by $X$ covers $\mathbb{R}^n$ and therefore $\Omega$.

We say that a point $x^0$ belongs to the interior of the convex hull of a set of points $X = \{x^1, \ldots, x^{n+1}\}$ if scalars $\alpha_j$ exist such that

$$x^0 = \sum_{j=1}^{n+1} \alpha_j x^j, \text{ where } \sum_{j=1}^{n+1} \alpha_j = 1 \text{ and } \alpha_j > 0 \text{ for } j = 1, \ldots, n+1. \tag{5.9}$$

This is denoted by $x^0 \in \operatorname{int}(\operatorname{conv}(X))$.

**Lemma 5.2.6** (Affine independence of initial points)**.** *If $X = \{x^1, \ldots, x^{n+1}\} \subset \mathbb{R}^n$ is an affinely independent set and if $x^0$ satisfies $x^0 \in \operatorname{int}(\operatorname{conv}(X))$, then all subsets of $n+1$ points in $\{x^0\} \cup X$ are affinely independent.*

*Proof.* For contradiction, suppose that the set $\{x^0\} \cup X \setminus \{x^{n+1}\}$ is not poised and therefore is affinely dependent. Then, there must exist scalars $\beta_j$ not all zero, and (without loss of generality) $x^n \in X$ such that

$$\sum_{j=0}^{n-1} \beta_j (x^j - x^n) = 0. \tag{5.10}$$

Replacing $x^0$ with (5.9) in the left-hand side above yields

$$\begin{aligned} 0 &= \beta_0 \left( \sum_{j=1, j\neq n}^{n+1} \alpha_j x^j + (\alpha_n - 1)x^n \right) + \sum_{j=1}^{n-1} \beta_j (x^j - x^n), \\ &= \sum_{j=1}^{n-1} (\beta_0 \alpha_j + \beta_j) x^j + \left( \beta_0(\alpha_n - 1) - \sum_{j=1}^{n-1} \beta_j \right) x^n + \beta_0 \alpha_{n+1} x^{n+1}. \end{aligned} \tag{5.11}$$

Since $X$ is poised, the vectors $\{x^1, \ldots, x^{n+1}\}$ are affinely independent; by definition of affine independence, the only solution to $\sum_{j=1}^{n+1} \gamma_j x^j = 0$ and $\sum_{j=1}^{n+1} \gamma_j = 0$ is $\gamma_j = 0$ for $j = 1, \ldots, n+1$. The sum of the coefficients from (5.11) satisfies

$$\sum_{j=1}^{n-1} (\beta_0 \alpha_j + \beta_j) + \beta_0(\alpha_n - 1) - \sum_{j=1}^{n-1} \beta_j + \beta_0 \alpha_{n+1} = \beta_0 \sum_{j=1}^{n+1} \alpha_j - \beta_0 = 0,$$

because $\sum_{j=1}^{n+1} \alpha_j = 1$. This means that all coefficients of $x^j$ in (5.11) are also equal to zero. Since $\alpha_{n+1} > 0$, the last term from (5.11) implies that $\beta_0 = 0$. Considering the remaining coefficients in (5.11), we conclude that $\beta_0 \alpha_j + \beta_j = 0$, which implies that $\beta_j = 0$ for $j = 1, \ldots, n-1$. This contradicts the assumption that not all $\beta_j = 0$. Hence, the result is proved. □

We now establish a simple set of points that produces conditional cuts that cover $\mathbb{R}^n$ and, therefore, the domain $\Omega$. First, we define an important concept used in many derivative-free algorithms.

**Definition 5.2.7** (Positive spanning set)**.** *(**?**, Theorem 2.3 (iii)) A positive spanning set is a set of vectors, $\{v^1, \ldots, v^r\} \subset \mathbb{R}^n$, such that for every $x \in \mathbb{R}^n$ there exists nonnegative coefficients, $\beta \in \mathbb{R}^r, \beta \geq 0$ such that $x = \sum_{j=1}^{r} \beta_j v^j$.*

**Lemma 5.2.8** (Initial points and coverage of $\Omega$)**.** *Let $X$ be an affinely independent set of $n+1$ points, let $x^0 \in \text{int}(\text{conv}(X))$, and let $W(X \cup \{x^0\})$ be defined as in (5.8). Then,*

1. *the vectors $(x^0 - x^j)$ for $j \in \{1, \ldots, n+1\}$ form a 'positive spanning set': that is any $x \in \mathbb{R}^n$ can be expressed as*

$$x = \sum_{j=1}^{n} \alpha_i (x^0 - x^j),$$

*with $\alpha_j \geq 0$ for all $j$, and $x^0$ has been chosen arbitrarily.*

2.

$$\bigcup_{\boldsymbol{i} \in W(X \cup \{x^0\})} \mathcal{U}^{\boldsymbol{i}} = \mathbb{R}^n.$$

*Proof.* Since $x^0 \in \operatorname{int}(\operatorname{conv}(X))$, there exist $\alpha_j > 0$ such that

$$0 = (\sum_{j=1}^{n+1} \alpha_j)(x^0 - x^0) = (\sum_{j=1}^{n+1} \alpha_j)x^0 - \sum_{j=1}^{n+1} \alpha_j x^j = \sum_{j=1}^{n+1} \alpha_j (x^0 - x^j), \tag{5.12}$$

where the second equality follows from (5.9). The existence of $\alpha_j > 0$ such that $\sum_{j=1}^{n+1} \alpha_j (x^0 - x^j) = 0$ implies that the vectors $\left\{x^0 - x^j : j \in \{1, \ldots, n+1\}\right\}$ are a positive spanning set (see Definition 5.2.7). Therefore any $x \in \mathbb{R}^n$ can be expressed as

$$x = \sum_{j=1}^{n} \alpha_j (x^0 - x^j),$$

with $\alpha_j \geq 0$ for all $j$.

We will show that any $x \in \mathbb{R}^n$ belongs to $\mathcal{U}^{\boldsymbol{i}}$ for some multi-index $\boldsymbol{i}$ containing $x^0$. By Lemma 5.2.6, every set of $n$ distinct vectors of the form $(x^0 - x^j)$ for $x^j \in X$ is a linearly independent set. Thus we can express

$$x - x^0 = \sum_{j=1, j\neq l}^{n+1} \lambda_j (x^0 - x^j), \tag{5.13}$$

for some $l \in \{1, \ldots n+1\}$. If $\lambda_j \geq 0$ for each $j$, then we are done, and $x \in \operatorname{cone}(x^0 - X \setminus \{x^l\})$.

Otherwise, choose an index $j'$ such that $\lambda_{j'}$ is the most negative coefficient on the right of (5.13) (breaking ties arbitrarily). Using (5.12), we can exchange the indices $l$ and $j'$ in (5.13) by observing that

$$\lambda_{j'}(x^0 - x^{j'}) = \frac{-\lambda_{j'}}{\alpha_{j'}} \left( \sum_{j=1, j\neq j'}^{n+1} \alpha_j (x^0 - x^j) \right).$$

Note that $\frac{-\lambda_{j'}}{\alpha_{j'}} \alpha_j > 0$ by (5.9), and we can rewrite (5.13) as

$$x - x^0 = \sum_{j=1, j\neq j'}^{n+1} \mu_j (x^0 - x^j), \tag{5.14}$$

with new coefficients $\mu_j$ that are strictly larger than $\lambda_j$:

$$\mu_j = \begin{cases} \lambda_j - \dfrac{\lambda_{j'}}{\alpha_{j'}} \alpha_j > \lambda_j, & j \neq l, j \neq j', \\ -\frac{\lambda_{j'}}{\alpha_{j'}} \alpha_j & j = l. \end{cases}$$

Observe that (5.14) has the same form as (5.13) but with coefficients $\mu_j$ that are strictly greater than $\lambda_j$. We can now define $\lambda = \mu$ and repeat the process. If there is some $\lambda_{j'} < 0$, the process will strictly increase all $\lambda_j$. Because there are only a finite number of subsets of size $n$, we must eventually have all $\lambda_j \geq 0$. Once $\lambda_{j'}$ has been pivoted out, it can reenter only with a positive value (like $\mu_l$ above), so eventually all $\lambda_j$ will be nonnegative. □

Lemma 5.2.8 ensures that any affinely independent set of $n+1$ points with an additional point in their interior will produce conditional cuts that cover $\mathbb{R}^n$. Figure 5.2 illustrates this for $n = 2$. An alternative set of $n+2$ points is

$$X = \{0, e_1, e_2, \ldots, e_n, -e\},$$

where $e_i$ is the $i$th unit vector and $e$ is the vector of ones. Larger sets, such as those of the form

$$X = \{0, e_1, -e_1, \ldots, e_n, -e_n\},$$

will similarly guarantee coverage of $\mathbb{R}^n$.

We note that the results in this section do not rely on $X$ or $\Omega$ being a subset of $\mathbb{Z}^n$. Therefore, the results are readily applicable to the case when $f$ has continuous and integer variables.

## 5.3 Proposed Algorithm and Convergence Analysis

We now present Algorithm 5.1 to identify global solutions to (5.1) under Assumption 5.1. Each iteration of Algorithm 5.1 involves updating the underestimator

**Algorithm 5.1:** Identifying a global minimizer of a convex objective on integer $\Omega$.

**Input:** A set of evaluated points $X^0 \subseteq \Omega$ satisfying $\left|W(X^0)\right| > 0$

Set $\hat{x} \in \arg\min_{x \in X^0} f(x)$, upper bound $u_0 \leftarrow f(\hat{x})$, and lower bound $l_0 \leftarrow -\infty$; $k \leftarrow 0$

**while** $l_k < u_k$ **do**

- Update: Update the piecewise linear program (PILP) using $W(X^k)$
- Lower Bound: Solve (PILP) and let its optimal value be $l_{k+1}$
- Next Iterate: Select a new trial point $x^{k+1} \in \Omega \setminus X^k$
- Evaluate $f(x^{k+1})$ and set $X^{k+1} \leftarrow X^k \cup \{x^{k+1}\}$
- **if** $f(x^{k+1}) < u_k$ **then**
  - Upper Bound: New incumbent $\hat{x} \leftarrow x^{k+1}$ and upper bound $u_{k+1} \leftarrow f(x^{k+1})$
- **else**
  - $u_{k+1} \leftarrow u_k$
- $k \leftarrow k + 1$

**Output:** $\hat{x}$, a global minimizer of $f$ on $\Omega$

of the form (PILP), and then minimizing the underestimator to update the best-known lower bound and also produce a point to be evaluated. This point replaces the current incumbent if it has a smaller function value. Algorithm 5.1 stops when the best-known lower bound equals the best-found function value.

Section 5.4 and Section 5.5 shows two approaches for modeling the underestimator and Section 5.5.1 highlights other details that are important for an efficient implementation of Algorithm 5.1. For example, the next point evaluated can be a solution of (PILP) but this needs not be the case.

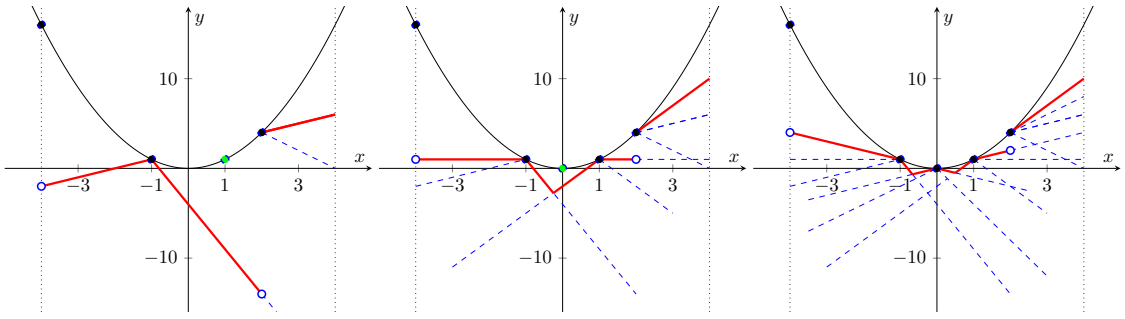
Note that (PILP) provides a valid lower bound for $f$ on $\Omega$. If $X \subseteq \Omega$ are points where $f$ has been evaluated, then $\min\{f(x) : x \in X\}$ is an upper bound for the minimum of $f$ on $\Omega$. Algorithm 5.1 terminates when the upper bound is equal to the lower bound provided by (PILP). We observe that Algorithm 5.1 produces a nondecreasing sequence of lower bounds provided that conditional cuts are not removed from (PILP); we show in Theorem 5.3.1 that this sequence of lower bounds will converge to the global minimum of $f$ on $\Omega$.

Algorithm 5.1 resembles a traditional outer-approximation approach (**???**) in that it obtains a sequence of lower bounds of (5.1) using an underestimator that is updated after each function evaluation. These function evaluations provide a nonincreasing sequence of upper bounds on the objective; when the upper bound equals the lower bound provided by the underestimator, the method can terminate with a certificate of optimality.

Algorithm 5.1 leaves open a number of important decisions concerning how (PILP) is formulated and solved and how the next iterate is selected. While we will discuss more involved options for addressing these concerns, a simple choice would be to add all new possible cuts and let the next iterate be a minimizer of (PILP). If this minimizer is not chosen, a (possibly difficult) separation problem may have to be solved to obtain a new iterate in Line 5, for example, when $\Omega$ is sparse. Although such choices can result in computational difficulties, these choices are useful for showing the behavior of Algorithm 5.1, which we do now. In Figure 5.3 we see three iterations of Algorithm 5.1 solving the one-dimensional problem

$$\text{minimize } f(x) = x^2 \quad \text{subject to } x \in [-4, 4], x \in \mathbb{Z}.$$

Black dots indicate interpolation points where $f$ has been previously evaluated, and green dots indicate the solution to (PILP) in each iteration. The solid red lines show the piecewise linear underestimator of the function. We observe that Lemma 5.2.1 can be strengthened for one-dimensional problems where condi-




Iter. 1: $l_k = -9, u_k = 1, \hat{x} = 1$ Iter. 2: $l_k = -0.27, u_k = 0, \hat{x} = 0$ Iter. 3: $l_k = 0, u_k = 0, \hat{x} = 0$

**Figure 5.3:** Illustration of Algorithm 5.1 minimizing $f(x) = x^2$ on $[-4, 4] \cap \mathbb{Z}$.

tional cuts underestimate convex $f$ at all points outside the convex hull of the points used to determine the corresponding secant function. (This is not true for $n > 1$.)

We now prove that Algorithm 5.1 identifies a global minimizer of $f$.

**Theorem 5.3.1** (Convergence of Algorithm 5.1)**.** *If* Assumption 5.1 *holds,* Algorithm 5.1 *terminates at an optimal solution* $x^*$ *of* (5.1) *in finitely many iterations.*

*Proof.* Algorithm 5.1 will terminate in a finite number of iterations because Assumption 5.1 ensures that $\Omega$ is finite and Line 5 ensures that $x^k$ is not a previously evaluated element of $\Omega$.

For contradiction, assume that Algorithm 5.1 terminates at iteration $k'$ with $f(\hat{x}) > f(x^*)$ for some $x^* \in \arg\min_{x \in \Omega} f(x)$. It follows from Line 8 that $x^* \notin X^{k'}$, because $f(x^*) < f(\hat{x})$. Lemma 5.2.5 ensures that the value of each conditional cut at $x^*$ is not larger than $f(x^*)$, which implies that $\eta(x^*) \leq f(x^*)$. Thus, the lower bound satisfies

$$l_{k'} \leq f(x^*) < f(\hat{x}) = u_{k'}.$$

Since $l_{k'} < u_{k'}$, Algorithm 5.1 did not terminate at iteration $k'$, giving a contradiction. Therefore, the result is shown. □

A special case of Theorem 5.3.1 ensures that Algorithm 5.1 terminates with a global solution of (5.1) when $x^k$ is an optimal solution of (PILP). As in most integer optimization algorithms, the termination condition $l_k = u_k$ may be met before $X^k = \Omega$; that is, Algorithm 5.1 needs not evaluate all the points in $\Omega$. Termination before $X^k = \Omega$ occurs when (PILP) is refined in Step 4 of Algorithm 5.1 and the lower bound at all points (and on the optimal value of (5.1)) is tightened. When the next iterate is chosen, the upper bound typically improves and convergence

occurs faster than enumeration. Yet, in the worst-case scenario, when cuts are unable to refine (PILP), the algorithm will evaluate all of $\Omega$. The lower bound at each point in $\Omega$ will be equal to the function value at that point, which will imply $l_k = u_k = \min_{x\in\Omega} f(x)$ for (PILP) (ensuring termination of the algorithm).

Algorithm 5.1 relies critically on the underestimator described by (PILP). Section 5.4 and Section 5.5 develops two approaches for formulating (PILP), and Section 5.5.1 discusses important details for efficiently implementing Algorithm 5.1. Section 5.5.2 combines these details in a description of our preferred method for solving (5.1), *SUCIL*.

## 5.4 Formulating (PILP) as an MILP Problem

We present two methods for encoding (PILP) and thereby obtain lower bounds for (5.1). The first approach formulates (PILP) as a mixed-integer linear program (MILP) using binary variables to indicate when a point $x$ is in $\mathcal{U}^{\boldsymbol{i}}$ for some multi-index $\boldsymbol{i}$. We show later in Section 5.5 that the resulting MILP is difficult to solve for even small problem instances. This motivates the development of the second approach, which directly builds an enumerative model of (PILP) in the space of the original variables only. First, we present the MILP based approach.

### 5.4.1 MILP Formulation

Formulating (PILP) as an MILP problem requires forming the secant function $m^{\boldsymbol{i}}(x) = (c^{\boldsymbol{i}})^\top x + b^{\boldsymbol{i}}$ corresponding to each multi-index $\boldsymbol{i} \in W(X)$. Since $m^{\boldsymbol{i}}$ is valid only in $\mathcal{U}^{\boldsymbol{i}}$ (see Lemma 5.2.5), we use binary variables to encode when $x \in \mathcal{U}^{\boldsymbol{i}}$. Explicitly, for each $\boldsymbol{i} \in W(X)$ and each $i_j \in \boldsymbol{i}$, our MILP model sets the binary variable $z^{i_j}$ to be 1 if and only if $x \in \operatorname{cone}(x^{i_j} - X^{\boldsymbol{i}})$. Although the forward implication can be easily modeled by using continuous variables $\lambda^{i_j}$, we must introduce additional binary variables $w^{i_j}$ for the reverse implication.

We now describe the constraints in the MILP model, some of which include "big-$M$" terms that can make the model difficult to solve numerically. The first set of constraints ensures that $\eta$ is no smaller than any of the conditional cuts that underestimate $f$:

$$\eta \geq (c^{\boldsymbol{i}})^\top x + b^{\boldsymbol{i}} - M_\eta \left(1 - \sum_{j=1}^{n+1} z^{i_j}\right), \quad \forall \boldsymbol{i} \in W(X), \tag{5.15}$$

where $M_\eta$ is a sufficiently large constant. By Lemma 5.2.4, we can add constraints to ensure that $x \in \Omega$ belongs to no more than one of the cones in $\mathcal{U}^{\boldsymbol{i}}$ for a given $\boldsymbol{i}$:

$$\sum_{j=1}^{n+1} z^{i_j} \leq 1, \ \forall \boldsymbol{i} \in W(X). \tag{5.16}$$

The following constraints define each point $x \in \Omega$ as a linear combination of the extreme rays of each cone$(x^{i_j} - X^{\boldsymbol{i}})$:

$$x = x^{i_j} + \sum_{l=1, l\neq j}^{n+1} \lambda_l^{i_j} \left(x^{i_j} - x^{i_l}\right), \quad \forall \boldsymbol{i} \in W(X), \ \forall i_j \in \boldsymbol{i}. \tag{5.17}$$

To indicate that $x \in \text{cone}(x^{i_j} - X^{\boldsymbol{i}})$, the following constraints enforce a lower bound of 0 on $\lambda$ when the corresponding $z^{i_j} = 1$:

$$\lambda_l^{i_j} \geq -M_\lambda \left(1 - z^{i_j}\right), \quad \forall \boldsymbol{i} \in W(X), \ \forall i_j, i_l \in \boldsymbol{i}, j \neq l, \tag{5.18}$$

where $M_\lambda$ is a sufficiently large constant. Next, we introduce the binary variables $w_l^{i_j}$ that are 1 when the corresponding variable $\lambda_l^{i_j}$ is nonnegative. The following constraints model the condition that $w_l^{i_j} = 0$ implies that the corresponding $\lambda_l^{i_j}$ takes a negative value:

$$\lambda_l^{i_j} \leq -\epsilon_\lambda + M_\lambda w_l^{i_j}, \quad \forall \boldsymbol{i} \in W(X), \ \forall i_j, i_l \in \boldsymbol{i}, j \neq l, \tag{5.19}$$

where $\epsilon_\lambda$ is a sufficiently small positive constant. The last set of constraints force at least one of the $w$ variables to be 0 if the corresponding $z$ is 0:

$$n z^{i_j} \leq \sum_{l=1, l\neq j}^{n+1} w_l^{i_j} \leq n - 1 + z^{i_j}, \quad \forall \boldsymbol{i} \in W(X), \ \forall i_j \in \boldsymbol{i}. \tag{5.20}$$

The full MILP model encoding of (PILP) is

$$\begin{aligned} &\underset{x,\lambda,z,w}{\text{minimize}} \ \eta \\ &\text{subject to (5.15)–(5.20)}, \\ &\qquad w_l^{i_j}, z^{i_j} \in \{0,1\}, \quad \forall l, j \in \{1,\ldots,n+1\}, l \neq j; \quad \forall \boldsymbol{i} \in W(X), \\ &\qquad x \in \Omega. \end{aligned} \tag{CPF}$$

### 5.4.2 Issues with MILP Formulation

The constants $M_\eta, M_\lambda$, and $\epsilon_\lambda$ must be chosen carefully in order to avoid numerical issues when solving (CPF). In early numerical results, we observed that taking large values for $M_\eta$ and $M_\lambda$ and small values for $\epsilon_\lambda$ resulted in numerical issues for the MILP solvers. In an attempt to remedy this situation, we derived cuts in which $c^{\boldsymbol{i}}$ and $b^{\boldsymbol{i}}$ are integer valued. We elaborate on these issues in this section.

*Derivation of Tolerances*

We show how one can compute sufficient values of $M_\eta$, $\epsilon_\lambda$, and $M_\lambda$ for the MILP model (CPF). We first show that if we choose these parameters incorrectly, then the resulting MILP model no longer provides a valid lower bound.

**Effect of an Insufficient $\epsilon_\lambda$ Value.** A large $\epsilon_\lambda$ or small $M_\lambda$ (or both) could result in an incorrect value of $z^{i_j} = 1$ for a point $x \notin \text{cone}(x^{i_j} - X^{\boldsymbol{i}})$, violating the implication of $z^{i_j} = 1$ and yielding an invalid lower bound on $f$. This is illustrated using a one-dimensional example in Figure 5.4. Similarly, one can encounter an invalid lower bound at an iteration if $M_\lambda$ is chosen to be smaller than required.

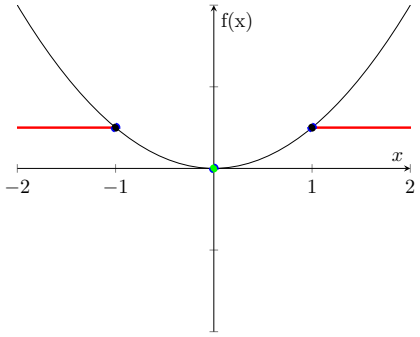



**Figure 5.4:** An example of false termination of Algorithm 5.1 using (CPF) when an insufficient value of $\epsilon_\lambda$ is used: $f(x) = x^2$ and interpolation points $-1$ and $1$ are used to form the secant shown in red colour. Any value of $\epsilon_\lambda > 0.5$ forces $z^{1_1} = z^{1_2} = 1$, activating the cut $\eta \geq 1$ at the optimal $x^* = 0$, resulting in $l_f^k = u_f^k = 1$.

**Bound on $M_\eta$** Let $l_f$ be a valid lower bound of $f$ on $\Omega$. With this lower bound, scalars $M_{\boldsymbol{i}}$ can be defined as

$$M_{\boldsymbol{i}} = \max_{x \in \Omega} \{(c^{\boldsymbol{i}})^\top x + b^{\boldsymbol{i}}\} - l_f, \ \forall \boldsymbol{i} \in W(X). \tag{5.21}$$

If $\Omega = \{x : l_x \leq x \leq u_x\}$, where $l_x, u_x \in \mathbb{R}^n$ are known, then we can set

$$M_{\boldsymbol{i}} = \sum_{h: c_j^{\boldsymbol{i}} < 0} c_h^{\boldsymbol{i}} l_x + \sum_{h: c_j^{\boldsymbol{i}} \geq 0} c_h^{\boldsymbol{i}} u_x + b^{\boldsymbol{i}} - l_f, \quad h = 1, \ldots, n, \ \forall \boldsymbol{i} \in W(X). \tag{5.22}$$

Then, we can either set individual values of $M_{\boldsymbol{i}}$ within each constraint (5.15), which would yield a tighter model, or use a single parameter,

$$M_\eta = \max_{\boldsymbol{i}} \{M_{\boldsymbol{i}}\}, \tag{5.23}$$

as shown in (CPF).

**Bounds on $\epsilon_\lambda$ and $M_\lambda$** Sufficient values of $M_\lambda$ and $\epsilon_\lambda$ are not easy to calculate as they depend on the rays generated at $x^{i_j}$, and the domain $\Omega$. We show next, that bounds on these values can be computed by solving a set of optimization problems. A sufficient value for $\epsilon_\lambda$ can be computed based on the representations of the $n+1$ hyperplanes formed using different combinations of $n$ points, $\forall \boldsymbol{i} \in W(X)$. We can use one of the representations of the hyperplane passing through points $X^i \setminus \{x^{i_j}\}$, $j = 1 \dots n+1$ to obtain bounds on $\epsilon_\lambda$, by solving an optimization problem for each $i_j \in \boldsymbol{i}$. Let $c^{\boldsymbol{i}_{i_j^-}}$ and $b^{\boldsymbol{i}_{i_j^-}}$ denote the solution of the following problem.

$$\begin{aligned} \underset{c,b}{\text{minimize}} \;& \|c\|_1 \\ \text{subject to } & c^\top x^{i_l} + b = 0, \quad l = 1, \dots, n+1,\ l \neq j, \\ & \|c\|_1 \geq 1, \\ & c \in \mathbb{Z}^n, b \in \mathbb{Z}. \end{aligned} \tag{P$-hyp$}$$

Problem (P$-hyp$) can be easily cast as an integer program by replacing $\|c\|_1$ by $e^\top y$, where $e = (1, \dots, 1)^\top$ is the vector or all ones, and the variables $y$ satisfy the constraints $y \geq c, y \geq -c$. Because, the hyperplane is generated using $n$ integer points, it can be shown that there exists a solution $c^{\boldsymbol{i}_{i_j^-}}$ and $b^{\boldsymbol{i}_{i_j^-}}$ that is nonzero and integral, as stated in the following proposition.

**Proposition 5.4.1.** *Consider a hyperplane $S = \{x : c^\top x + b = 0\}$ such that $c$ and $b$ are integral. The Euclidean distance between an arbitrary point $\hat{x} \in \mathbb{Z}^n \setminus S$, and $S$, is greater than or equal to $\frac{1}{\|c\|_2}$.*

*Proof.* The Euclidean distance between a point $\hat{x}$ and $S$ is the 2-norm of the projection of the line segment joining an arbitrary point $w \in S$ and $\hat{x}$, on the normal passing through $\hat{x}$, and can be expressed as

$$\frac{|c^\top \hat{x} + b|}{\|c\|_2}. \tag{5.24}$$

By integrality of $c, b$ and $\hat{x}$, and since $\hat{x} \notin S$, $|c^\top \hat{x} + b| \geq 1$, and the result follows. □

Proposition 5.4.1 can be used directly to get a sufficient value of $\epsilon_\lambda$ as follows.

$$\epsilon_\lambda = \min_{\boldsymbol{i} \in W(X),\, i_j \in \boldsymbol{i}} \frac{1}{\|c^{\boldsymbol{i}_{i_j^-}}\|_2}. \tag{5.25}$$

The bound can be tightened using the following optimization problem.

$$\begin{aligned} \underset{x}{\text{minimize}} \;& \frac{|(c^{\boldsymbol{i}_{i_j^-}})^\top x + b^{\boldsymbol{i}_{i_j^-}}|}{\|c^{\boldsymbol{i}_{i_j^-}}\|_2} \\ \text{subject to } & |(c^{\boldsymbol{i}_{i_j^-}})^\top x^{i_l} + b^{\boldsymbol{i}_{i_j^-}}| \geq 1, \\ & x \in \Omega. \end{aligned} \tag{P-$\epsilon$}$$

If we denote by $\epsilon_\lambda^{\boldsymbol{i}_{i_j^-}}$, the optimal value of (P-$\epsilon$), then the following is a sufficient value of $\epsilon_\lambda$.

$$\epsilon_\lambda = \min_{\boldsymbol{i} \in W(X),\, i_j \in \boldsymbol{i}} \epsilon_\lambda^{\boldsymbol{i}_{i_j^-}}. \tag{5.26}$$

Similarly, if we maximize the objective in (P-$\epsilon$), and denote the optimal value by $M_\lambda^{\boldsymbol{i}_{i_j^-}}$ we get a sufficient value for $M_\lambda$ as follows.

$$M_\lambda = \max_{\boldsymbol{i} \in W(X),\, i_j \in \boldsymbol{i}} M_\lambda^{\boldsymbol{i}_{i_j^-}}. \tag{5.27}$$

**No-Good Cuts and Stronger Objective Bounds.** For our initial experiments, we tried setting arbitrarily small values for $\epsilon_\lambda$ instead of obtaining the best possible values by solving a set of optimization problems as elaborated above. The problem with having such values of $\epsilon_\lambda > 0$ too small is that it allows us to ignore the cuts at the interpolation points, $x^{(k)}$, causing a violation of the bound, $\eta \geq f^{(k)}$, for $x = x^{(k)}$ in the MILP model, resulting in repetition of iterates in the algorithm. We fixed this by adding the following valid inequality to the MILP model.

$$\eta \geq f^{(k)} - M\|x^{(k)} - x\|_1.$$

We can write this inequality as a linear constraint, by introducing a binary representation of the variables, $x$, requiring $nU$ binary variables, $\xi_{ij}$, $i = 1, \ldots, n$, $j = 1, \ldots, U$, where $n$ is the dimension of our problem, and $0 \leq x \leq U$. With this representation, we obtain

$$x_i = \sum_{j=1}^{U} i\xi_{ij}, \quad 1 = \sum_{j=1}^{U} \xi_{ij}, \quad \xi_{ij} \in \{0, 1\},$$

and write the $\eta$-constraint equivalently as

$$\eta \geq f^{(k)} - M \sum_{i=1}^{n} \left( \sum_{j:\xi_{ij}^{(k)}=0} \xi_{ij} + \sum_{j:\xi_{ij}^{(k)}=1} \left(1 - \xi_{ij}\right) \right).$$

One can spawn fewer binary variables using binary variables in the following way,

$$x_i = \sum_{j=1}^{\lfloor U_i \rfloor} 2^j \cdot \xi_{ij}, \quad \xi_{ij} \in \{0, 1\},$$

however, we do not elaborate more on this as the MILP approach is not scalable in this context.

One can then show, for example, that $1/\|c^{\boldsymbol{i}}\|_2$ is a valid lower bound for $\epsilon_\lambda$, and similar tight bounds can be derived for $M_\lambda$. With these tighter constants,

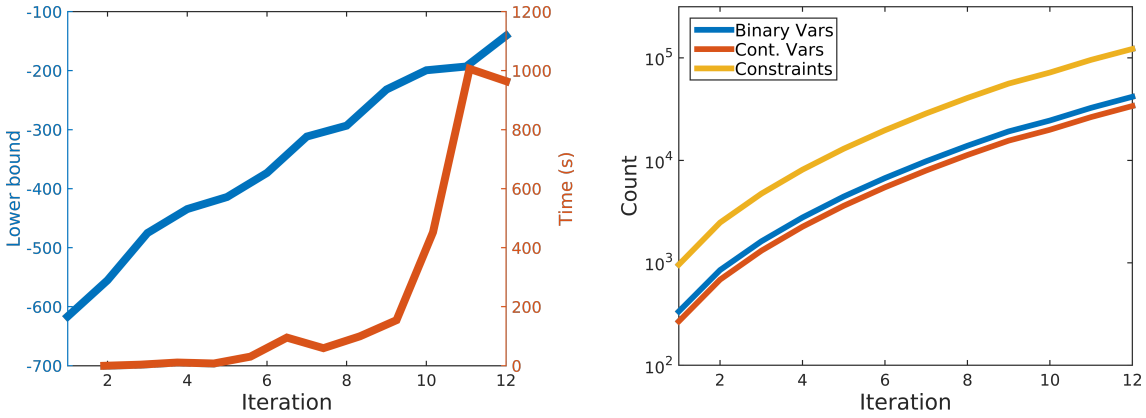



**Figure 5.5:** Characteristics of the first 12 instances of (CPF) generated by Algorithm 5.1 minimizing the convex quadratic `abhi` on $\Omega = [-2, 2]^3 \cap \mathbb{Z}^3$. Left shows the lower bound and solution time (mean of five replications and maximum and minimum times are also shown); right shows the number of binary and continuous variables and constraints. For further details of these 12 MILP models, see Table 5.1.

some numerical issues were resolved. Yet, the growth in the number of constraints in (CPF) prevented its application to problems with $n \geq 3$.

Initial versions of the MILP model (CPF) resulted in large times to solution. Figure 5.5 shows the behavior of Algorithm 5.1—when adding all possible cuts when updating (PILP) and choosing the next iterate be a minimizer of (PILP)—when minimizing the convex quadratic function `abhi` (defined in Table A.4 of Appendix A) on $\Omega = [-2, 2]^3 \cap \mathbb{Z}^3$. We note that the variations in CPU time are consistent over five repeated runs and vary by less than 2.4% for the last two iterations. The solution time of MILP solvers depends critically on implementation features, including presolve operations, node selection rules, and branching preferences. After the additional set of cuts (constraints) are introduced in iteration 12 of this problem instance, the MILP solver was able to solve the problem in slightly less time than the previous iteration. (Such occurrences are not rare in MILPs: time to solution is not strictly increasing in problem size.) Overall, we find that the growth in CPU time is due to the increasing number of conditional cuts and the associated explosion in the number of binary and continuous variables. This trend appears to limit the applicability of the MILP approach. Note that the global minimum of `abhi` on $[-2, 2]^3 \cap \mathbb{Z}^3$ has not yet been encountered when the MILPs become too large to solve. (The iteration 13 MILP instance was not solved in 30 minutes.)

**Table 5.1:** Characteristics of the first 12 instances of (CPF) generated by Algorithm 5.1 minimizing the convex quadratic function `abhi` on $\Omega = [-2, 2]^3 \cap \mathbb{Z}^3$. (CPF) instances are generated by AMPL and solved by CPLEX; times are the mean of five replications.

| $k$ | *sHyp* | *LB* | *UB* | *time* | *simIter* | *nodes* | *bVars* | *cVars* | *cons* | $\hat{x}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 20 | -616.3 | 79.9 | 0.1 | 374 | 0 | 335 | 268 | 960 | [2; 2; −2] |
| 2 | 52 | -555.1 | 79.9 | 3.9 | 13,180 | 8,089 | 847 | 685 | 2,466 | [2; 2; −1] |
| 3 | 100 | -475.2 | 44.7 | 10.9 | 31,423 | 8,555 | 1,615 | 1,310 | 4,724 | [2; 1; −2] |
| 4 | 172 | -434.4 | 44.7 | 7.3 | 19,267 | 1,728 | 2,767 | 2,247 | 8,110 | [1; 2; −2] |
| 5 | 276 | -413.9 | 19.1 | 30.8 | 68,264 | 5,874 | 4,431 | 3,600 | 13,000 | [2; 1; −1] |
| 6 | 418 | -373.1 | 19.1 | 95.1 | 84,031 | 7,933 | 6,703 | 5,447 | 19,676 | [1; 2; −1] |
| 7 | 611 | -311.7 | 19.1 | 59.6 | 83,102 | 5,440 | 9,791 | 7,957 | 28,749 | [2; −2; −2] |
| 8 | 866 | -293.2 | 19.1 | 99.6 | 86,318 | 3,933 | 13,871 | 11,273 | 40,736 | [1; 1; −2] |
| 9 | 1,196 | -232.0 | 19.1 | 154.2 | 84,440 | 4,568 | 19,151 | 15,564 | 56,248 | [1; 1; −1] |
| 10 | 1,532 | -199.5 | 19.1 | 452.3 | 235,473 | 6,400 | 24,527 | 19,933 | 72,042 | [2; −2; −1] |
| 11 | 2,038 | -192.9 | 19.1 | 1,006 | 387,491 | 9,686 | 32,623 | 26,512 | 95,826 | [2; −1; −2] |
| 12 | 2,605 | -140.9 | 19.1 | 964.3 | 455,939 | 29,279 | 41,695 | 33,884 | 122,477 | [1; −1; −2] |

Table 5.1 shows the size of the MILP model at each iteration and the computational effort required for solving it. The column $k$ refers to the iteration of Algorithm 5.1, *sHyp* denotes the number of secants (i.e., $|W(X)|$), and *LB* and *UB* give the lower and upper bound on $f$ on $\Omega$, respectively. We show the computational effort needed to solve each MILP instance via *time*, the mean solution time (in seconds) for 5 replications; *simIter*, the number of simplex iterations; and *nodes*, the number of branch-and-bound nodes explored by the MILP solver. The size of each MILP instance (after presolve) is shown in terms of *bVars*, the number of binary variables; *cVars*, the number of continuous variables; and *cons*, the number of constraints. Table 5.1 also shows the optimal solution $\hat{x}$ of each MILP instance. These experiments were performed by using CPLEX (v.12.6.1.0) on a 2.20 GHz, 12-core Intel Xeon computer with 64 GB of RAM. For this small problem we see that the size of the MILP instance grows exponentially as the iterations proceed, which results in an exponential growth in solution time as illustrated in Figure 5.5. The iteration 13 MILP instance was not solved after 30 minutes.

## 5.5 Enumerative Approach

Whereas the MILP model from Section 5.4 encodes information about every conditional cut in a single model, this section considers an alternative approach of updating the value of $\eta(x)$ for each $x \in \Omega$ as new conditional cuts are encountered. After the information from a new secant function is used to update $\eta(x)$, the secant is discarded.

Ordering the finite set of feasible integer points as $\left\{x^1, x^2, \ldots, x^{|\Omega|}\right\}$, our approach maintains and updates a vector of bounds

$$\left[\eta(x^1), \eta(x^2), \ldots, \eta(x^{|\Omega|})\right]^\top \in \mathbb{R}^{|\Omega|}, \tag{5.28}$$

where $\eta(x^j)$ is the value of (PILP) when $x = x^j$. The value of $\eta(x^j)$ is initialized to $-\infty$; and as each secant is constructed, $\eta(x^j)$ is set to the maximum of its current value and the value of the conditional cut at $x^j$. This procedure is described in Algorithm 5.2. Since the important information about each conditional cut will be stored in $\eta(x)$, the secants defining each cut do not need to be stored. (Not storing secants comes at a cost of storing $\eta(x)$ for each $x \in \Omega$, which may be prohibitive if $|\Omega|$ is large.)

If $\eta_k(x)$ is the value of the underestimator (5.28) at iteration $k$, then solving each instance of (PILP) corresponds to looking up $\underset{j\in\{1,\ldots,|\Omega|\}}{\arg\min}\, \eta_k(x^j)$ (breaking ties arbitrarily). Similarly, termination of Algorithm 5.1 requires testing only that $\min_{j\in\{1,\ldots,|\Omega|\}} \eta_k(x^j) \geq u_k$.

Note that when solving (5.1), updating $\eta(x)$ for all $x \in \Omega$ is unnecessary. Rather, one needs to update $\eta(x)$ only at points that could possibly be a global minimum of $f$ on $\Omega$. When $f$ is evaluated at $x^{k+1}$ and a multi-index $\boldsymbol{i} \in W(X^k \cup x^{k+1})$ is encountered that is not in $W(X^k)$, we update the lower bound only at points in $\mathcal{U}^{\boldsymbol{i}}$ that are also in

$$\Omega_k = \{x \in \Omega \setminus X^k : \eta_k(x) < u_k\}. \tag{5.29}$$

That is, we update $\eta_k(x)$ for points in $\mathcal{U}^{\boldsymbol{i}}_k = \Omega_k \cap \mathcal{U}^{\boldsymbol{i}}$ for each newly encountered $\boldsymbol{i}$.

**Algorithm 5.2:** Routine for updating lower bound for $f$ at each point in $\Omega$.

```
Function UpdateEta(X^i, b^i, c^i, U^i_k, η(x)):
    for i_k ∈ i do
        for j = 1, ..., |Ω| do
            if x^j ∈ cone(x^{i_k} − X^i) ∩ U^i_k then
                η(x^j) ← max(η(x^j), (c^i)^⊤ x^j + b^i)
```

### 5.5.1 Other Implementation Details

The enumerative approach of maintaining the value of the underestimator $\eta(x)$ described in Section 5.5 avoids many of the computational pitfalls of the

MILP model discussed in Section 5.4. Below, we discuss additional computational enhancements that lead to an efficient implementation of Algorithm 5.1 in conjunction with Algorithm 5.2.

*Checking Whether $X^{\boldsymbol{i}}$ Is Affinely Independent and Whether $x \in \mathcal{U}^{\boldsymbol{i}}$*

We now describe a numerically efficient representation of $\text{cone}(x^{i_j} - X^{\boldsymbol{i}})$ for $i_j \in \boldsymbol{i}$. Given an affinely independent set of $n+1$ points, $X^{\boldsymbol{i}}$, for each $i_j \in \boldsymbol{i}$ we define a secant function satisfying

$$(c^{i_j})^\top x^{i_l} + b^{i_j} = 0, \text{ for all } i_l \in \boldsymbol{i}, i_l \neq i_j, \text{ and} \tag{5.30}$$

$$(c^{i_j})^\top x^{i_j} + b^{i_j} > 0. \tag{5.31}$$

Only one such secant exists for each $i_j \in \boldsymbol{i}$; however, the representation of this secant is not unique since $(c^{i_j}, b^{i_j})$ are obtained by solving an underdetermined system of equations. Given $(c^{i_j}, b^{i_j})$ satisfying (5.30) and (5.31), we define the corresponding halfspace,

$$H^{i_j} = \{x : (c^{i_j})^\top x + b^{i_j} \leq 0\}. \tag{5.32}$$

We now show that $\text{cone}(x^j - X^{\boldsymbol{i}})$, defined in (5.5), can be represented as the intersection of $n$ such halfspaces.

**Lemma 5.5.1** (Set equality). *For an affinely independent set $X^{\boldsymbol{i}}$, $\text{cone}(x^{i_j} - X^{\boldsymbol{i}}) = F^{i_j} = \bigcap_{i_l \neq i_j} H^{i_l}$ for each $i_j \in \boldsymbol{i}$.*

*Proof.* Let $\boldsymbol{i}$ be given and $i_j \in \boldsymbol{i}$ fixed. We first show that $\text{cone}(x^{i_j} - X^{\boldsymbol{i}}) \subseteq F^{i_j}$ by showing that an arbitrary $x \in \text{cone}(x^{i_j} - X^{\boldsymbol{i}})$ satisfies (5.32) for each $i_l \in \boldsymbol{i}, i_l \neq i_j$. Given $(c^{i_l}, b^{i_l})$ satisfying (5.30) and (5.31), then using the definition (5.5) yields

$$\begin{aligned}
(c^{i_l})^\top x + b^{i_l} &= (c^{i_l})^\top \left( x^{i_j} + \sum_{k=1, k\neq j}^{n+1} \lambda_k (x^{i_j} - x^{i_k}) \right) + b^{i_l}, \\
&= (c^{i_l})^\top x^{i_j} + b^{i_l} + \sum_{k=1, k\neq j}^{n+1} \lambda_k (c^{i_l})^\top x^{i_j} - \sum_{k=1, k\neq j}^{n+1} \lambda_k (c^{i_l})^\top x^{i_k}, \\
&= 0 + \sum_{k=1, k\neq j}^{n+1} \lambda_k \left( (c^{i_l})^\top x^{i_j} + b^{i_l} \right) - \sum_{k=1, k\neq j}^{n+1} \lambda_k \left( (c^{i_l})^\top x^{i_k} + b^{i_l} \right), \\
&= 0 - \sum_{k=1, k\neq j}^{l-1} \lambda_k \left( (c^{i_l})^\top x^{i_k} + b^{i_l} \right) - \lambda_l \left( (c^{i_l})^\top x^{i_l} + b^{i_l} \right) - \sum_{k=l+1, k\neq j}^{n+1} \lambda_k \left( (c^{i_l})^\top x^{i_k} + b^{i_l} \right), \\
&= -\lambda_l \left( (c^{i_l})^\top x^{i_l} + b^{i_l} \right) \leq 0,
\end{aligned}$$

where we have used (5.30) in the last three equations. The final inequality holds because $\lambda_l \geq 0$ by (5.5) and $(c^{i_l})^\top x^{i_l} + b^{i_l} > 0$ by (5.31). Because $i_l$ is arbitrary, it follows that any $x$ in $\text{cone}(x^{i_j} - X)$ is also in $F^{i_j}$.

We now show that $F^{i_j} \subseteq \text{cone}(x^j - X^{\boldsymbol{i}})$ by contradiction. If $x \notin \text{cone}(x^{i_j} - X^{\boldsymbol{i}})$ for a set of $n+1$ poised points $X^{\boldsymbol{i}}$, then $x$ can be represented as $x^{i_j} + \sum_{l=1, l\neq j}^{n+1} \lambda_l \left(x^{i_j} - x^{i_l}\right)$ only with some $\lambda_l < 0$. Thus, (5.32) is violated for some $l$, and hence $x \notin F^{i_j}$. □

Lemma 5.5.1 gives a representation of each $\text{cone}(x^{i_j} - X^{\boldsymbol{i}})$ involving $n$ halfspaces that differs from $\text{cone}(x^{i_l} - X^{\boldsymbol{i}})$ for $i_l \in \boldsymbol{i}$, $i_l \neq i_j$ in only one component. Therefore, we can represent $\mathcal{U}^{\boldsymbol{i}}$ via only $n+1$ halfspaces. We efficiently calculate these halfspaces by utilizing the QR factorization $\left[Q^{\boldsymbol{i}}\ R^{\boldsymbol{i}}\right] = \left[\bar{X}^{\boldsymbol{i}}\ e\right]^\top$. If $R^{\boldsymbol{i}}$ has positive diagonal entries, then the multi-index $\boldsymbol{i}$ corresponds to an affinely independent set $X^{\boldsymbol{i}}$. The coefficients in each $(c^{i_j}, b^{i_j})$ can be obtained by updating $Q^{\boldsymbol{i}}$, $R^{\boldsymbol{i}}$ by deleting the corresponding column from $\left[\bar{X}^{\boldsymbol{i}}\ e\right]^\top$. The sign of $(c^{i_j}, b^{i_j})$ can be changed in order to ensure that (5.31) holds.

*Approximating $W(X^k \cup \{x^{k+1}\})$*

The use of $\eta_k(x)$ to store the lower bound at each $x \in \Omega_k$ allows us to avoid encoding all secants in $W(X^k)$. After $f$ has been evaluated at a new point $x^{k+1}$, constructing the tightest possible underestimator in $\eta_k$ requires considering multi-indices $\boldsymbol{i}$ in $W(X^k \cup \{x^{k+1}\})$ that contain $x^{k+1}$. (Combinations not containing $x^{k+1}$ have already been considered in previous iterations.) While not storing secants is significantly more computationally efficient than encoding and storing all secants in $W(X^k)$, it still results in checking the affine independence of prohibitively many sets of $n+1$ points. For example, if $\left|X^k\right| = 100$ and $n = 5$, over 75 million QR factorizations must be performed, as discussed in Section 5.5.1.

Therefore, as an alternative, we seek a small, representative subset of multi-indices of $W(X^k)$ by identifying a subset of points that will yield the best conditional cuts.

**Definition 5.5.2.** *Let $\bar{W}_k$ be the set of multi-indices in $W(X^k)$ that define the largest lower bound at some point in $\Omega_k$ (defined in (5.29)). That is, $\bar{W}_k = \{\boldsymbol{i} : \exists x \in \Omega_k$ such that $\eta_k(x) = m^{\boldsymbol{i}}(x)\}$. We denote the* generator *set of points as $G^k = \{x^j : \exists \boldsymbol{i} \in \bar{W}_k$ such that $j \in \boldsymbol{i}\}$.*

Hence, $G^k$ contains points that define $\eta_k(x)$ for at least one $x \in \Omega_k$. Using $W(G^k)$ in place of $W(X^k)$ does relax (PILP), yet the lower bounding property of

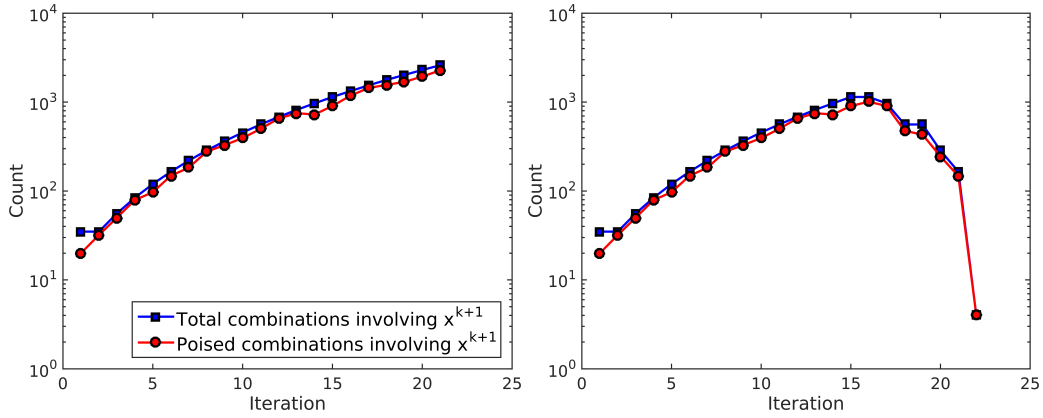



**Figure 5.6:** Number of total combinations and affinely independent combinations that include $x^{k+1}$ in $W(X^k)$ (left) and $W(G^k)$ (right) when minimizing `quad` on $\Omega = [-4, 4]^3 \cap \mathbb{Z}^3$.

(PILP) still remains. We show below that this change does not affect the finite termination property of Algorithm 5.1 provided at least one cut is added for every new $x^{k+1}$.

Figure 5.6 compares the growth of the number of subsets of indices that must be considered when determining whether a multi-index $\boldsymbol{i}$ is affinely independent or not when using Algorithm 5.1 to minimize `quad` (Table A.4 in Appendix A) on $\Omega = [-4, 4]^3 \cap \mathbb{Z}^3$. Preliminary numerical experiments showed that although a high percentage of all combinations in $W(X^k \cup x^{k+1})$, which involve the new iterate $x^{k+1}$ at an iteration $k$, are affinely independent, only a small fraction of these actually update the lower bound at any point in $\Omega_k$ (we elaborate more on this in Section 5.7).

*Selecting $x^{k+1}$*

Early experiments with our algorithm showed that it spent many early iterations evaluating points at the boundary of $\Omega$. Although Section 5.2.3 provides a method for ensuring that all $x \in \Omega$ are bounded by at least one conditional cut, the solution to (PILP) is often at the boundary of $\Omega$. Rather than moving so far from a candidate solution, we consider a trust-region approach to keep iterates close to the current incumbent. As long as we maintain a lower bound for $f$ on $\Omega$, the convergence proof in Theorem 5.3.1 does not depend on $x^{k+1}$ being the global minimizer of our lower bound.

In practice, we use an infinity-norm trust region and set the minimum trust-region radius, $\Delta_{\min}$ to 1. At iteration $k$, the maximum radius that must be considered is $\max_{x,y \in \Omega_k, x \neq y} \|x - y\|_\infty$.

### 5.5.2 The *SUCIL* Method

We now present the *SUCIL* —secant underestimator of convex functions on the integer lattice— method for obtaining global solutions to (5.1) under Assumption 5.1. The algorithm using the trust-region step is shown in Algorithm 5.3. We observe that Algorithm 5.3 maintains a valid lower bound $\eta_k(x)$ at every point, $x \in \Omega_k$, and that the trust-region mechanism ensures that the algorithm terminates only when the lower bound equals the best observed function value.

We note that $G^k$ may not be a subset of $G^{k+1}$, because $\Omega_k$ can contain fewer points as the upper and lower bounds on $f$ are improved. However, the following generalization of Theorem 5.3.1 ensures that Algorithm 5.3 still returns a global minimizer of (5.1).

**Theorem 5.5.3** (Convergence of Algorithm 5.3)**.** *If* Assumption 5.1 *holds and if* $W(G^k)$ *includes at least one cut for every* $x \in \Omega_k$*, then* Algorithm 5.3 *terminates at an optimal solution* $x^*$ *of* (5.1) *in finitely many iterations.*

*Proof.* Algorithm 5.3 will terminate in a finite number of iterations because $\Omega$ is bounded and Line 16 ensures that $x^k$ is not a previously evaluated element of $\Omega$. Because $W(G^k) \subset W(X^k)$, it follows that $\eta_k(x)$ is a valid lower bound for $f$ on $\Omega$, and the trust-region mechanism in Line 18 ensures that Algorithm 5.3 terminates only if $l_{k+1} = u_k$. Therefore, the result is shown. □

Both Algorithm 5.1 and Algorithm 5.3 have a worse than exponential time complexity because they solve an MILP problem (indirectly, in Algorithm 5.3). For integer programming algorithms like branch-and-bound, typically a weak bound is given by complete enumeration. Our algorithms may have to enumerate all points in $\Omega$ when our secants are ineffective. For example, if $\Omega = \{0, 1\}^n$, the described approaches must evaluate all $2^n$ points. A noteworthy expensive computation in Algorithm 5.3 is the construction of the set $G^k$ (see Line 5) that requires finding sets of $n$-sized combinations of points in $X^k$ at each iteration $k$. This can amount to $O((|X^k|)!/n!(|X^k| - n)!)$ such sets. Different tasks and checks then have to be performed corresponding to each element in $G^k$. For example, we compute a QR factorization ($O(n^3)$) for each element in $G^k$ to check if a point lies in $\mathcal{U}^i$ and update the lower bound if the point belongs to $\mathcal{U}^i$.

**Algorithm 5.3:** The *SUCIL* method for convex MIDFO.

**Input:** A set of evaluated points $X^0 \subseteq \Omega : \bigcup_{\boldsymbol{i} \in W(X^0)} \mathcal{U}^{\boldsymbol{i}} = \mathbb{R}^n$ and trust-region radius lower bound $\Delta_{\min} \geq 1$

Set $\hat{x} \in \arg\min_{x \in X^0} f(x)$, upper bound $u_0 \leftarrow f(\hat{x})$, $\Omega_0 \leftarrow \Omega$, and $k \leftarrow 0$.

Initialize lower bounding function $\eta_{-1}(x) \leftarrow -\infty$ for all $x \in \Omega$; set lower bound $l_0 \leftarrow -\infty$.

**while** $l_k < u_k$ **do**

  Update:

  Generate $G^k$ (according to Definition 5.5.2) using $X^k$ .

  **for** $\boldsymbol{i} \in W(G^k)$ **do**

    Compute $QR$ factors: $[Q, R] \leftarrow \text{qr}([e\ X^{\boldsymbol{i}}])$.

    **if** $X^{\boldsymbol{i}}$ *is affinely independent* **then**

      Find $c^{\boldsymbol{i}}, b^{\boldsymbol{i}}$ and form set $\mathcal{U}_k^{\boldsymbol{i}} \leftarrow \Omega_k \cap \mathcal{U}^{\boldsymbol{i}}$ using $QR$ factors.

      Update look-up: $\eta_k \leftarrow$ `UpdateEta(`$X^{\boldsymbol{i}}, b^{\boldsymbol{i}}, c^{\boldsymbol{i}}, \mathcal{U}_k^{\boldsymbol{i}}, \eta_{k-1}$`)`; see Algorithm 5.2.

  Lower Bound:

  $l_{k+1} \leftarrow \min_{x \in \Omega_k} \eta_k(x)$ from look-up table.

  **if** $l_{k+1} = u_k$ **then**

    **break**

  Next Iterate:

  Update $\Omega_k \leftarrow \{x \in \Omega \setminus X^k : \eta_k(x) < u_k\}$ .

  **if** $\{x \in \Omega_k : \|x - \hat{x}\| \leq \Delta_k\} = \emptyset$ **then**

    Increase trust-region radius: $\Delta_k \leftarrow \Delta_k + 1$ until $\{x \in \Omega_k : \|x - \hat{x}\| \leq \Delta_k\} \neq \emptyset$ .

  **else**

    Set $x^{k+1} \in \arg\min_{x \in \Omega_k : \|x - \hat{x}\| \leq \Delta_k} \eta_k(x)$.

    Evaluate $f(x^{k+1})$ and set $X^{k+1} \leftarrow X^k \cup \{x^{k+1}\}$.

    **if** $f(x^{k+1}) < u_k$ **then**

      Upper Bound:

      New incumbent $\hat{x} \leftarrow x^{k+1}$ and upper bound $u_{k+1} \leftarrow f(x^{k+1})$.

      Increase trust-region radius $\Delta_{k+1} \leftarrow \Delta_k + 1$ .

    **else**

      No progress: $u_{k+1} \leftarrow u_k$ and reduce trust-region radius $\Delta_{k+1} \leftarrow \max\left\{\Delta_{\min}, \frac{\Delta_k}{2}\right\}$.

  $k \leftarrow k + 1$

**Output:** $\hat{x}$, a global minimizer of $f$ on $\Omega$

| Method | $X$ in (PILP)? | $x^{k+1}$ =? |
|---|---|---|
| *SUCIL* | $G^k$ | $\arg\min_{x\in\Omega_k:\|x-\hat{x}\|_\infty\leq\Delta_k} \eta_k(x)$ |
| *SUCIL-noTR* | $G^k$ | $\arg\min_{x\in\Omega_k} \eta_k(x)$ |
| *SUCIL-ideal1* | $X^k$ | $\arg\min_{x\in\Omega\setminus X^k} f(x)$ |
| *SUCIL-ideal2* | $G^k$ | $\arg\min_{x\in\Omega\setminus X^k} f(x)$ |

**Table 5.2:** Description of how *SUCIL* versions choose $X$ in (PILP) and the next iterate $x^{k+1}$ (breaking ties in arg min arbitrarily). $G^k$ is defined in Definition 5.5.2, and $X^k$ is all points evaluated before iteration $k$.

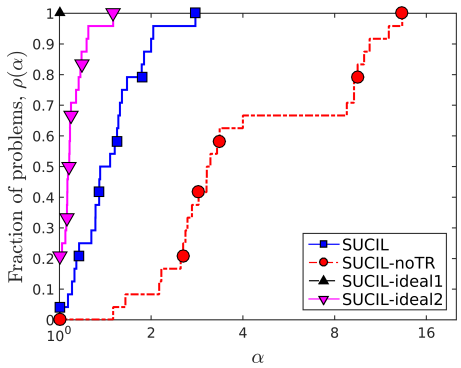



**Figure 5.7:** Performance profiles for *SUCIL*s. Convergence measured by number of function evaluations before a method terminates with a certificate of global optimality.

## 5.6 Numerical Experiments

We now describe numerical experiments performed on multiple versions of *SUCIL*; see Table 5.2. These methods differ in how $x^{k+1}$ is selected and in the set of points used within (PILP). The last two methods are idealized because they assume access to the true function value at every point in $\Omega_k$. They are included in order to provide a best-case performance for a *SUCIL* implementation. In the numerical experiments to follow, we set $\Delta_{\min} \leftarrow 1$ in Algorithm 5.3 and use an infinity-norm trust region. All *SUCIL* instances begin by evaluating the starting point $\bar{x}$ and $\{\bar{x} \pm e_1, \ldots, \bar{x} \pm e_n\}$ ensuring a finite lower bound at every point in $\Omega$.

Below, we compare *SUCIL* implementations with a direct-search method, *DFLINT* (**?**); a model-based method, *MATSuMoTo* (**?**); and a hybrid method, *NOMAD* (**?**). We tested the default nonmonotone *DFLINT* in MATLAB, as well as the monotone version, denoted *DFLINT-M*. We tested the default C++ version of *NOMAD* (v.3.9.0) as well as the same version with *DISABLE MODELS* set to true, denoted *NOMAD-dm*; the rest of the settings are default. *MATSuMoTo* is a surrogate-model toolbox explicitly designed for computationally expensive, black-box, global optimization problems. Since *MATSuMoTo* has a restarting mechanism that ensures that any budget of function evaluations will be exhausted, we input the optimal objective function value to *MATSuMoTo* and allowed it to run (and make as many restarts as required) until the global optimal value was identified. The default settings were used: surrogate models using cubic radial basis functions, sampling at the minimum of the surrogate, and us-

ing an initial symmetric Latin hypercube design. We performed 20 replications of *MATSuMoTo* for each problem instance. We show the results from the replication that takes a number of function evaluations less than but closest to the median value for each problem instance. A common starting point is given to all methods; the starting point for the problems `entropy`, `infnorm`, `maxq`, `mxhilb` and `onenorm` is the global minimizer. A maximum function evaluation limit of 1,000 is set for all the methods when there are more than 1000 points in $\Omega$.

We perform numerical experiments minimizing the convex objectives in Table A.4 in Appendix A on the domains $[-K, K]^n \cap \mathbb{Z}^n$ for $K = 4, 10, 20$ for $n \in \{3, 4\}$ and $K = 4$ for $n = 5$ to yield 112 problem instances. (The last row of Table 5.1 shows $|\Omega|$ for these test problems.) While we choose to test our method on domains $\Omega$ defined only by bound constraints on $x$ variables, general convex constraints can be also be accommodated by adding a small mechanism to check the feasibility of the new iterate. Of note is the `KLT` function that generalizes the example function by **?** that shows how coordinate search methods can fail to find descent. The function by **?** is itself a modification of the Dennis-Woods function (**?**), is strongly convex, and points $x$ along the line $x_1 = \cdots = x_n$ satisfy $f(x) < f(x \pm \epsilon e_i)$ for all $i$ and for all $\epsilon > 0$. The problems `CB3II`, `CB3I`, `LQ`, `maxq`, and `mxhilb` were introduced by **?** and also used by **?**. These five problems are either summation or maximization of generalizations of simple convex functions, constructed by extending or chaining nonsmooth convex functions or making smooth functions nonsmooth. The function `LQ` takes a global minimum at any $x \in [0, 1]^n \cap \mathbb{Z}^n$ that does not have zeros in consecutive coordinates. For example, for $n = 3$, the points $[0, 1, 0]^\top, [0, 1, 1]^\top, [1, 0, 1]^\top, [1, 1, 0]^\top$, and $[1, 1, 1]^\top$ are optimal but $[0, 0, 0]^\top, [0, 0, 1]^\top$, and $[1, 0, 0]^\top$ are not. Problem `reciprob` is a slightly modified version of the foundational case of concave reciprocal functions (**?**, Exercise 4.4.10) whose convexity is difficult to establish through Hessian computations. Problem `entropy` (**?**, Section 6.4) has applications in nuclear magnetic resonance analysis. If there are relatively few points in $\Omega$ and the time required to evaluate $f$ is small (as for our test instances), one could argue that an enumerative procedure itself could solve the problem in a reasonable time. However, we use these instances to thoroughly examine the behavior that might be seen on expensive-to-evaluate black-box functions. Therefore, we compare methods using performance profiles (**?**) that are based on the number of function evaluations required to satisfy the respective convergence criterion. For each method $s$, $\rho_s(\alpha) = \frac{|\{p \in P : r_{p,s} \leq \alpha\}|}{|P|}$, for a scalar $\alpha \geq 1$, $P$ is the collection of problems, and

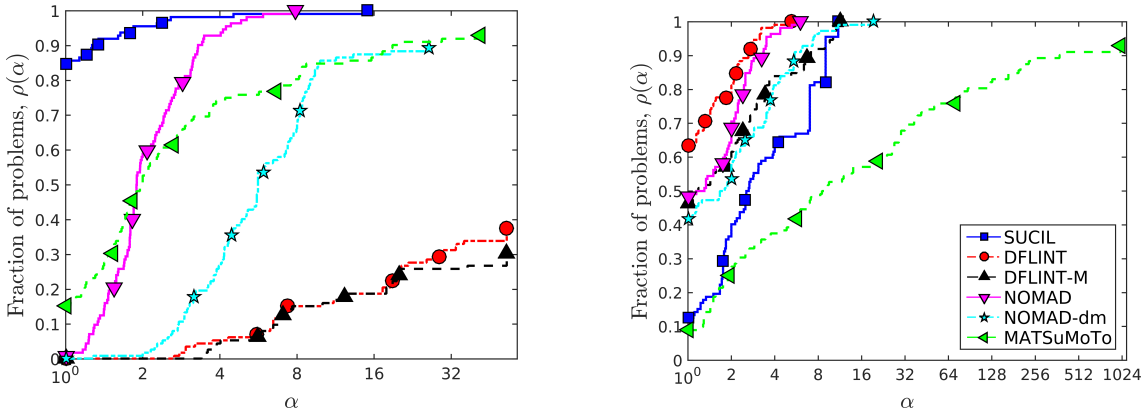



**Figure 5.8:** Performance profiles of different methods solving 112 problem instances. Left compares the number of evaluations until a method terminates; right compares the number of evaluations before a method first evaluates a global minimizer. Performance for each solver on each problem instance can be found in Tables A.1–A.3 in Appendix A.

$r_{p,s} = \frac{N_{p,s}}{\min_{s \in S}\{N_{p,s}\}}$ is the performance ratio. We consider two measures of $N_{p,s}$: (1) the number of function evaluations before a method $s$ terminates on a problem $p$ and (2) the number of function evaluations taken by method $s$ to evaluate a global minimizer on problem $p$.

Figure 5.7 compares the number of evaluations required for four implementations of *SUCIL* to terminate (with a certificate of optimality) on the set of test problems. While *SUCIL-ideal1* is no slower than any other implementation on all the test problems, it is not a realistic method in that it evaluates points based on their known function values. *SUCIL* requires no more than three times the evaluations as *SUCIL-ideal1* for the set of test problems. We do observe that using a trust region in *SUCIL* is a significant advantage. For many of the problems considered, *SUCIL-noTR* spent many function evaluations in the corners of $\Omega$.

As a point of comparison with the results in Figure 5.7, a different estimate of the number of function evaluations (or primitive directions explorations) required for the proof of optimality for our instances can be seen in Table 5.1, in columns corresponding to $n \in \{3, 4, 5\}$ and $k = 4$. As evident from the results in Tables A.1–A.3, our method incurs a remarkably low number of function evaluations for many problems, which can be attributed to exploitation of convexity and subsequent formation of the underestimators, as explained in Section 5.1. Note that for many problems (with fixed $n$), the number of function evaluations to prove optimality does not grow as $K$ increases.

We now analyze the performance of *SUCIL* compared with the other methods. Figure 5.8 (left) shows the performance profiles for methods to terminate on the 112 test problems; Figure 5.8 (right) compares the number of function evaluations required before each method first evaluates a global minimizer. This comparison is nontrivial because each solver has its own design considerations and notions of local optimality. Also, the other solvers do not assume convexity of the problem or exploit it. Hence, our results merely demonstrate that *SUCIL* provably converges to a global optimum and uses fewer function evaluations to certify global optimality because of its exploitation of convexity. Figure 5.8 shows that our algorithm requires the least number of function evaluations for more than 80% of the instances and provides a global optimality certificate, in addition. In reaching the global optimal solution quickly, however, DFLINT wins for more than 60% of the instances. Although *SUCIL* is not particularly designed to greedily descend to the global optimum, it is still competitive with the rest of the methods on this front.

Next, we examine the behavior of various solvers using data profiles with convergence test as described by **?**. Figure 5.9 (left) shows the data profiles for $\tau = 0$. We observe that *DFLINT* performs slightly better compares to other solvers within the function evaluation range $[16, 64]$, however, *DFLINT-M* and *SUCIL* are quite close. For an accuracy level $\tau = 0.5$ (right, Figure 5.9), most of the solvers perform identically, except *MATSuMoTo*.

While our approach finishes in seconds for many problems, this does not hold for all problems. For example, the $n = 5$, $K = 4$ instance of `reciprob` required approximately 1 minute over its 44 function evaluations and `mxhilb` required 5 minutes for its 144 function evaluations. As the problem domain grows, so can the computational requirements. The worst-case problem was `onenorm` for $n = 5$, $K = 4$, which required approximately three days to complete. (Nearly all of this time was spent constructing and using the various secants to update the lower bound $\eta$.)

## 5.7 Discussion

The order of results in this paper tells the story of how we arrived at the implementation of *SUCIL*. We first attempted to classify where linear interpolation models provide lower bounds for convex functions, yielding the results in Section 5.2; we then proved that such linear functions can underlie a convergent

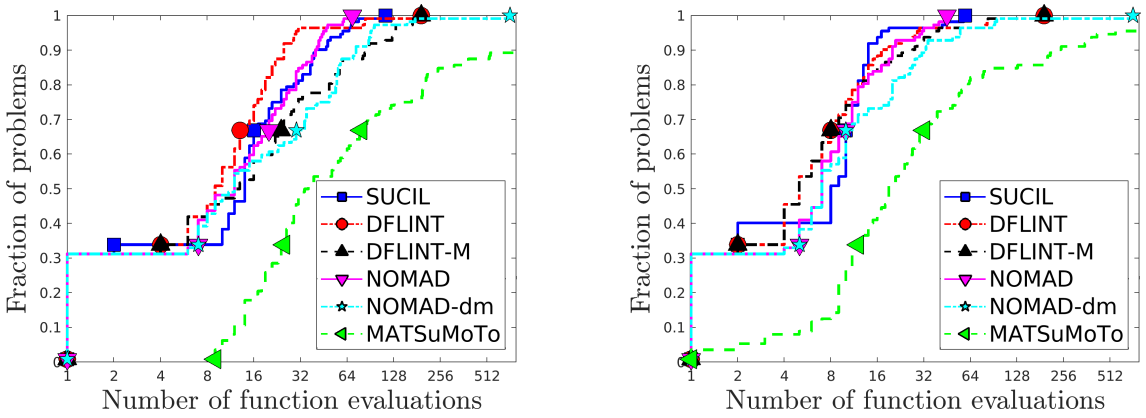



**Figure 5.9:** Data profiles of different methods solving 112 problem instances. Left profile corresponds to $\tau = 0$ and right corresponds to $\tau = 0.5$.

algorithm, as in Section 5.3. We initially modeled the secants and the conditions in which they are valid as an MILP problem, as in Section 5.4. After observing that the number of variables in the MILP model was larger than the number of points in the domain $\Omega$, we were motivated to develop the enumerative model in Section 5.5.

*Unconditional cuts*

Our computational developments expose a number of fundamental challenges for integer derivative-free optimization. The complexity of our piecewise linear model (PILP) is made worse by the fact that each secant function is valid only in the union of $n+1$ cones $\mathcal{U}^i$, resulting in conditional cuts. We note that it may not be possible to derive *unconditional cuts*, that is, cuts that are valid in the whole domain $\Omega$. For example, we might initially consider secants interpolating a convex $f$ at the $n+1$ points $x \in \mathbb{Z}^n$ and $x \pm e_i \in \mathbb{Z}^n$, where for every $i$ we can choose either $+$ or $-$. Such points form a unit simplex that has no integer points in its interior. Consequently, one might suspect that the resulting cut is valid everywhere in $\Omega$. However, the following example shows that the resulting cut is *not* unconditionally valid. Consider $f(x) = x_1^2 - x_1x_2 + x_2^2$ and the set of points $\{[1,1]^\top, [0,1]^\top, [1,0]^\top\}$. It follows that $f(x) = 1$ at these points, and hence the unique interpolating secant function is the constant function, $m(x) = 1$. Now consider the point $x = [0,0]^\top$ for which $f(x) = 0$, which is not underestimated by $m(x) = 1$.

*Binary domains*

Another limitation of our method is that it will have to evaluate all feasible points when $\Omega = \{0,1\}^n$ (a pure binary domain) for convergence because no point in $\Omega \backslash X^{\boldsymbol{i}}$ belongs to $\mathcal{U}^{\boldsymbol{i}}$, where $X^{\boldsymbol{i}}$ is an arbitrary affinely independent set of $n+1$ points in $\{0,1\}^n$. (Complete enumeration would also be required by any method searching a discrete 1-neighborhood for a binary problem.)

*Number of secant evaluations*

In Figure A.1 in Appendix A we show the number of function evaluations needed to first evaluate a global minimizer and the additional number of evaluations used to prove it is a global minimizer for some of our test functions. As is common, the effort required to certify optimality can be significantly larger than the cost of finding the optimum. In terms of number of function evaluations required, the proof of optimality is even more time consuming. Because the iterations where $X^k$ or $G^k$ is large require checking many potential secant functions, in *SUCIL* the computational cost of iterations can differ by orders of magnitude as the algorithm progresses.

Although our method provides a practical iterative way to check sufficiency of a set of points (optimality conditions) for a given convex instance, each iteration involves construction and evaluation of a large number of combinations of different $n+1$ points, which limits the scalability of Algorithm 5.3 in solving instances of higher dimensions. Yet, in our numerical experiments, we observe that only a small fraction of the total cuts evaluated are useful. We call $(c^{\boldsymbol{i}}, b^{\boldsymbol{i}})$ an *updating* cut at an iteration $k$ if there exists an $x \in \Omega_k$ such that $m^{\boldsymbol{i}}(x) > \eta_k(x)$, that is, a cut that improves the lower bound at at least one $x \in \Omega_k$. In addition, if $m^{\boldsymbol{i}}(x) \geq u_k$, we call it a *pruning* cut. A pruning cut helps eliminate points to be considered in the next iteration ($\Omega_{k+1}$). Figure 5.10 shows the number of updating and pruning cuts generated per iteration of *SUCIL* when minimizing `quad` on $[-4,4]^3 \cap \mathbb{Z}^3$.

*Separation of useful cuts*

The fact that few cuts prune a point or update the lower bound at any point where the minimum could be suggests that there may be some way to exclude a large set of multi-indices from consideration, possibly yielding dramatic computational savings. For example, it can be shown that it is not necessary to consider any multi-index that corresponds to a set of points that contain another point in

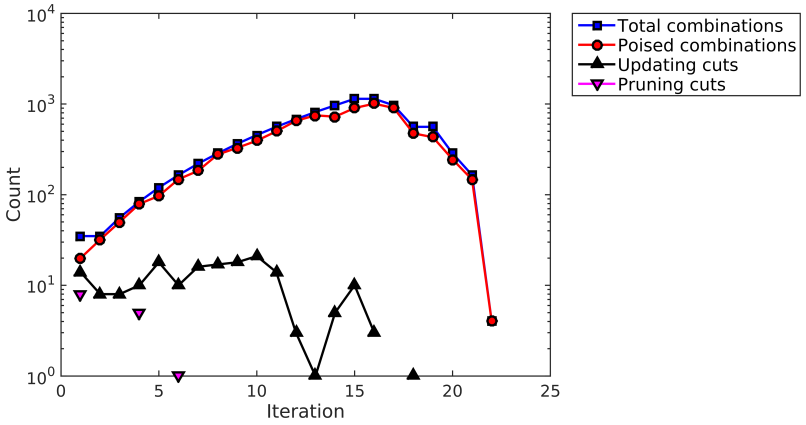



**Figure 5.10:** Number of total and affinely independent combinations of $n+1$ points, the secant functions that update, and the secant functions that prune at least one point when minimizing `quad` on $\Omega = [-4,4]^3 \cap \mathbb{Z}^3$ using *SUCIL*. (Markers are removed when there is no updating or no pruning cut in an iteration.)

its convex hull. Unfortunately, we are unaware of any efficient approach for generating the subsets of $X$ that contain $n+1$ point and that do not contain any other point in $X$ in their convex hull.

Ideally, we would like to evaluate only the combinations that yield updating or pruning cuts. However, this approach requires the solution of a separate problem that we believe is especially hard to solve. Even the following simpler problem of finding a pruning cut at a given candidate point seems difficult.

**Problem 1.** *Given a point $\bar{x} \in \mathbb{Z}^n$, a set of (integer) points $X$ where $f$ has been evaluated, and scalar $u$, find a cut that prunes $\bar{x}$. That is, find a multi-index $\boldsymbol{i}$ such that $\bar{x} \in \mathcal{U}^{\boldsymbol{i}}$ and $(c^{\boldsymbol{i}})^\top \bar{x} + b^{\boldsymbol{i}} \geq u$, and $(c^{\boldsymbol{i}}, b^{\boldsymbol{i}})$ solves (5.3), or show that no such multi-index $\boldsymbol{i}$ exists.*

If we choose a small subset $\bar{X}^k$ of $X^k$ to form $W(\bar{X}^k)$, the *SUCIL* algorithm can end up using a large number of function evaluations to obtain a certificate of optimality, which is not desirable because we target expensive derivative-free functions. The reason is that points are evaluated that would be ruled worse than optimal if secants were built by using all combinations of points in $X^k$. This situation occurred when setting $\bar{X}^k$ to be a random subset of $X^k$, a subset of the points closest to $\hat{x}$, or a subset of points with best function values. Using $G^k$ avoids discarding too many points from $X^k$; but we observe a significant increase in $\left|W(G^k)\right|$, and thus we incur heavy computational costs during some iterations. The wall clock time required per iteration for solving instances of dimension less than 5 in our setup is not significant, but we present the same for 5-dimensional instances

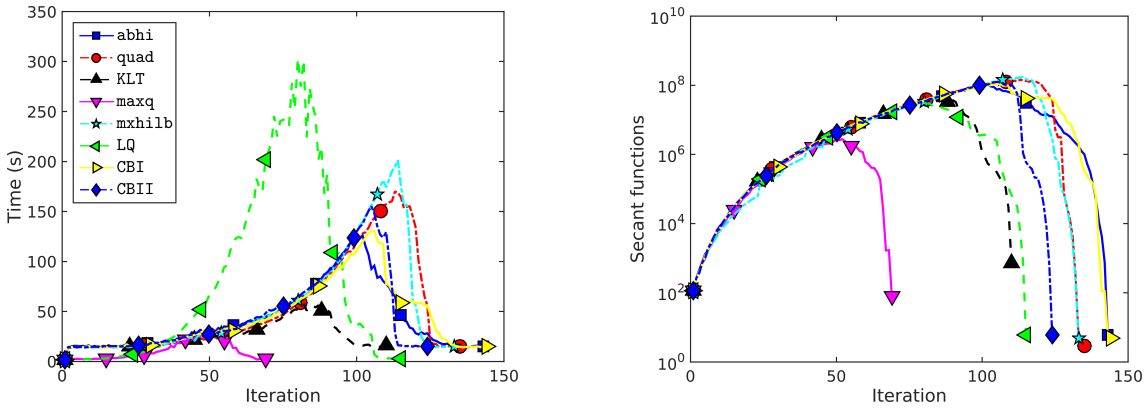



**Figure 5.11:** Wall-clock time recorded and number of secants constructed per iteration of *SUCIL* for 8 convex test problems on $\Omega = [-4, 4]^5 \cap \mathbb{Z}^5$.

using *SUCIL* on a 96-core Intel Xeon computer with 1.5 TB of RAM. The complexity of our approach is better quantified by counting the number of combinations of points (or potential secants) considered at iteration $k$. Using $G^k$, we typically produce a strict subset of all possible combinations in such a way that the size of $W(G^k)$ decreases during the later iterations. This is shown in Figure 5.11: the number of secants added per iteration for all 5-dimensional test instances using *SUCIL*. Once $\Omega_k$, the number of points with $\eta(x)$ less than $f(\hat{x})$, starts decreasing, so do $G^k$ and $\left|W(G^k)\right|$. In general, it is difficult to predict when the number of combinations (or the wall clock time curve) would be at the peak, but we suspect this peak will be worse as $n$ increases, by both the size and the iteration number where it occurs. This limits the applicability of the current implementation of *SUCIL* on higher-dimensional problems.

We also note that the storage requirements for the enumerative model may be prohibitive, even for moderate problem sizes. For example, an array storing the value of $\eta(x)$ as an 8-byte scalar for all $x \in \Omega = [-10, 10]^{10} \cap \mathbb{Z}^{10}$ would require over 200 GB of storage.

*Sufficient sets ($X^*$) that prove optimality*

Again, since nearly all cuts in $W(G^k)$ do not update $\eta(x)$ at any point in $\Omega_k$ (see Figure 5.10), we believe there may be some approach for intelligently selecting points from $X^k$ using their geometry, their function values, and distance from the best available iterate that will rule some multi-indices $\boldsymbol{i}$ as unnecessary to consider. We did attempt to identify minimal sets of points that were necessary

for *SUCIL* to certify optimality for a variety of $n = 2$ test cases, but no general rule was apparent.

We are interested in the sets $X^* \subset \Omega$ of points, such that the lower bound derived from the model (PILP) is equal to $f(x^*)$, which proves optimality. We call such a set of points a sufficient set of interpolation points (clearly, a necessary condition is that at least one $x^* \in X^*$).

Unfortunately, it seems nontrivial to construct low cardinality sufficient sets, even in the $n = 2$ case, as the following two examples illustrate. We consider the two quadratics in $\mathbb{R}^2$, `quad` and `abhi`, and construct sufficient sets to prove optimality of $x^* = (2, 2)$, the optimal solution for both these functions. Figure 5.12 depicts the continuous contours of the functions `quad` and `abhi` that comprise at least one integer point. The optimal $x^* = (2, 2)$ is depicted as a green solid circle. A set which includes the optimal, the black square dots and any two of the magenta colored diamond dots, is sufficient for the convergence of Algorithm 5.1. Similarly, we present one such sufficient set for `abhi` in the same figure. Algorithm 5.1 converges for this problem if the interpolation set includes the green solid circle and the black square dots.

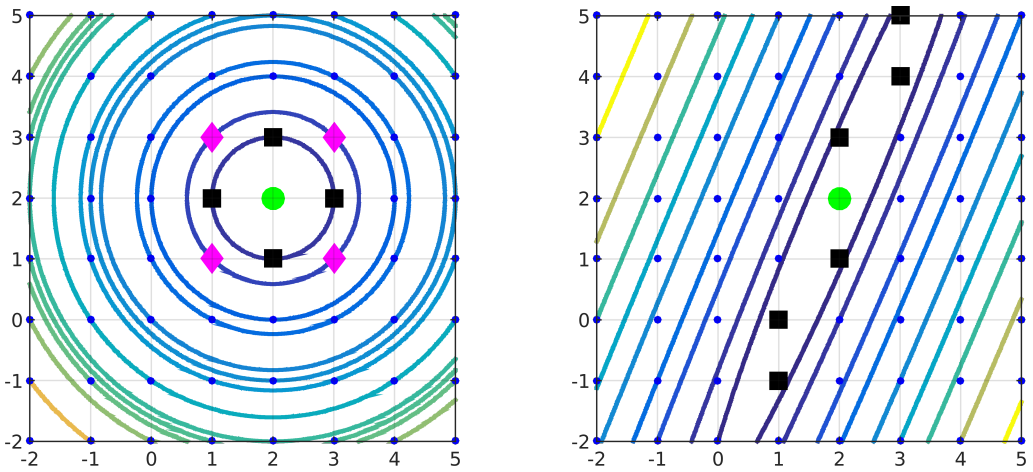



**Figure 5.12:** Points sufficient for Algorithm 5.1 to prove optimality on `quad` (left) and `abhi` (right).

The examples illustrate that the shape of the sufficient set depends strongly on the function that is minimized. It is not clear how to construct a minimal set of interpolation points that prove optimality.

In addition, the size of a minimum cardinality optimal set may not be a polynomial function of $n$. AS already mentioned earlier, lets take the example `quad`,

where just by observation of Figure 5.12, one can infer that all the $2^n$ points, including the unique optimum, are *necessary* for $f = \sum_i^n x_i^2$ when $x \in \{0, 1\}^n$ i.e. Algorithm 5.3 will not terminate unless every point $x \in \Omega$ has been evaluated.

*A different underestimator model*

Perhaps it is possible to construct a convex, continuous piecewise-linear underestimator to remedy many of these issues. We are unaware of how such a model would be constructed and updated when new points and function values become available. Moreover, we conjecture that it is not possible to build a general convex underestimation model for a convex derivative-free function just by using the first order information, that is, just the pairs of points and their function values.

Ultimately, we believe further insights are yet to be discovered that will facilitate better algorithms for minimizing convex functions on integer domains.

# Chapter 6

# Conclusions and Future Work

We study MINLPs, a general class of mathematical optimization problems, that are theoretically difficult to solve due to the presence of nonlinear functions and integer constrained variables. We address two cases of MINLPs. In the first case, the functions involved in the objective and the constraints are smooth and differentiable. We present share-memory parallel algorithms for these MINLPs and analyze their performance via extensive computational experiments on benchmarking instances. In the second case, we study MIDFO problems. Here, mathematical descriptions of the functions in the MINLPs and their derivatives are not available. In addition, every function evaluation is expensive. Under certain assumptions, we present methods similar to outer approximation for solving MIDFO problems to global optimality using only function evaluations.

Chapter 2 presents parallel algorithms for convex MINLP. We consider the serial and parallel versions of four algorithms: NLP-BB with sharing of branching information between threads (*mcbnbSRel*), QG with extra linearizations and parallelization using Minotaur's own branch-and-cut implementation (*mcqgHyb*), QG with branch-and-cut implementation of CPLEX MILP solver running in opportunistic mode (*lstoaO*), and OA with CPLEX MILP solver using all solutions from the solution pool of CPLEX (*oaSol*). We analyze the performance of parallel algorithms for scalability and compare them amongst each other and with two other parallel MINLP solvers, FSCIP (**?**) and SHOT (**?**). The goal is to study the effects of various algorithmic components and implementations on the performance of algorithms.

We observe that all solvers benefit from parallelization, although none obtains perfect scalability. We also observe that OA (that uses an external MILP solver) performs better than the QG algorithm of Minotaur that has its own

branch-and-cut implementation and lacks several key MILP features. Overall, QG using callbacks of an MILP solver outperformed other algorithms. Parallel QG (with enhanced linearization schemes shown by **?**) turned out to be the next best algorithm. Parallel extensions of the NLP-BB and QG report improvement upon their sequential counterparts by more than 40% using up to 16 threads. The speedup is more prominent for difficult instances. As the number of nodes processed increased by only about 60%, the proposed parallel algorithms can be made more compact and opportunistic. Nevertheless, advanced MILP techniques seem to impact the MINLP algorithms to a large extent and are the best alternatives subject to the availability of a good MILP solver.

In Chapter 3, we study anomalies in parallel tree-search algorithms, in particular, detrimental anomalies in NLP-BB and QG algorithms for convex MINLPs. The existence of detrimental anomalies may cause the parallel algorithms to perform worse than the sequential algorithm. We present a simple but practical unambiguous branching scheme that preserves the nondetrimental property of the tree-search for NLP-BB. Moreover, for QG, we extend the notion of unambiguity to cut generation, which is essential to make the overall algorithm nondetrimental. Our numerical results show that such guarantees are practically attainable in state-of-the-art parallel MINLP algorithms by using unambiguous components. We also achieve deterministic reproducible runs using these algorithmic components.

In Chapter 4, we propose a multi-start branch-and-estimate heuristic for nonconvex MINLP, that approximately solves each node of the branch-and-bound tree using parallelism. Each node in this tree is a nonconvex NLP. We experiment with five randomized schemes to generate initial points for solving the nonconvex NLPs using a local NLP solver, to find good solutions to nonconvex NLPs. We compared different schemes in our heuristic amongst themselves and compared the best settings with two global solvers, SCIP and Couenne, and with a convex MINLP algorithm, the NLP-BB from Minotaur. Our computational results show that the multi-start heuristic performs better than NLP-BB in terms of solution quality. Compared to the global solver, the solution quality obtained by our heuristic is comparable, while it fares better in terms of the time taken. Overall, the multi-start method seems promising in terms of obtaining good solutions heuristically for nonconvex problems, and parallelism helps in exploring more in the same amount of computational time.

In Chapter 5, we first present a way to obtain an underestimator of a derivative-free function, assuming that the objective function is convex. Underestimators for MIDFO problems are rare, and so are provably convergent global optimization algorithms, even for the convex case. We present a globally convergent algorithm based on our underestimator. We formulate the underestimator as a piecewise linear MILP model and keep tightening it using function evaluations until the algorithm converges. We present two algorithmic implementations, one that exploits an MILP solver and another which we call SUCIL, that heuristically maintains the bounds at each point in the domain. Our computational results show we require much fewer function evaluations compared to other MIDFO solvers like DFLINT, MATSuMoTo etc. This is because we exploit convexity in our algorithm. Although, our algorithmic implementations are mainly for the proof of concept, there is a huge scope of improvement on this front. The first issue is the sensitivity of the MILP model to tolerances, and second is the design of SUCIL algorithm, due to which it can not be scaled up for instances of larger sizes.

Next, we highlight a few promising research directions that stem out of the work presented in this thesis.

**Single-Tree OA for Nonconvex MINLP** Our computational results show that *lstoa*, the QG algorithm that uses lazy cuts callback functionality of MILP solvers, exhibits the best performance. This algorithm is also called single-tree OA by **??**. This is presumably due to the advanced MILP techniques like conflict graphs, presolving, cuts, etc. in MILP solvers that eventually outweigh the other MINLP algorithms like QG and NLP-BB where the MINLP solver manages its own LP and NLP based trees. Recently, on similar lines as *lstoa*, **?** have solved a mixed-integer quadratic bilevel problem, where bilevel-specific structure of the problem is exploited to ultimately reduce the problem to a single-level convex MIQCQP, which is then solved using multi- and single-tree OA. On some instances, the single-tree algorithm outperformed the multi-tree algorithm. Leveraging on the current effectiveness of MILP solvers, one can solve nonconvex MINLPs, especially, MIQCQPs, in a spatial branch-and-bound like framework, using the callback functionalities of an advanced MILP solver. In such a possible algorithm, a deviation from the existing (convex) ones would be in creation of the LP nodes within the MILP solver when one would branch on continuous variables. Also, the child nodes would require more than just changing the bounds of variables.

For example, envelops of bilinear terms (**?**) are refined significantly when bounds change. One way to accomplish such refinements is by using branching callbacks provided by the MILP solver. Another issue is that the cuts using the integer solutions (for example, the outer-approximation cuts) obtained in the MILP tree may no longer be valid. So, other specific (convex) reformulations based on the MIQCQP structure of the problem might have to be explored, that yield good valid inequalities.

**Multi-Level Distributed- and Shared-Memory Parallel Frameworks** Most state-of-the-art parallel MINLP algorithms (**???**) typically focus on only one level of parallelism, and are efficient only up to a certain number of processors (**?**). One can design multi-level parallel frameworks that distributes data and/or tasks to individual processing units depending upon some known estimate of the scalability limits at each level. If available, shared-memory parallelism can be readily used at some lower level. The distribution of available processing units can be statically or dynamically decided based on the scalability of tasks at each level. A direct simple extension could be a parallel subtree-level NLP-BB for convex MINLPs. One can solve at the second level, different NLPs of the subtree using shared-memory parallelism (using dynamic or some preassigned number $k$ of) processors up to which it scales well, and at the first level distribution of subtrees could be done. Another different idea is to design an MILP-based branch-and-bound for nonconvex MINLPs. MILP solvers have been observed to scale up to 4 or 8 processors (**?**) in similar contexts, hence the first level distribution of MILP subtrees can be done based on this number.

**Novel Models and Relaxations for MIDFO** There are several theoretical and computational challenges open on MIDFO front; we mention a few immediate extensions. State-of-the-art methods address the integrality in MIDFO problems either using local searches (**?**), or integer directions (**?**) or using surrogate models (**?**). Surrogate models could be convex or nonconvex, and represent an approximate mathematical function based on the values evaluated at some points. Within MIDFO algorithms, these models are optimized using appropriate MINLP algorithms. When the surrogate model is nonconvex, it is typically not solved to global optimality because solving nonconvex MINLPs is difficult. But one can solve such embedded nonconvex MINLPs using global MINLP solvers directly or by exploiting the special structures in nonconvex functions using the

techniques from nonconvex MINLP. Problem specific heuristics can also be used. In case the integer variables are unrelaxable, as in Chapter 5, continuous relaxations based approaches to find a valid bounding relaxation do not work. But one can explore novel bounding methods for problems with unrelaxable integer variables.

# Appendix A

# Test Problems and Numerical Results for MIDFO

Tables A.1–A.3 contain detailed numerical results for the interested reader. Note that some solvers do not respect the given budget of function evaluations. We have used a different stopping criterion for *MATSuMoTo*: it is set to stop only when a point with the optimal value has been identified. Also, although the global minimum has been provided as the starting point to all the solvers for `entropy`, `infnorm`, `onenorm`, `maxq` and `mxhilb`, the *MATSuMoTo* solver uses its initial symmetric Latin hypercube design. The last row of Table 5.1 in Section 5.1 shows $|\Omega|$ for these problems.

Table A.4 shows the details of the convex problems used for benchmarking the MIDFO solvers.

**Table A.1:** Number of function evaluations taken by solvers for 7 of the 16 base problems and various values of $n$ and $K$. For *MATSuMoTo*, the nearest value less than the median of the number of the evaluations of 20 replications is chosen.

| Problem | *SUCIL* | *DFLINT* | *DFLINT-M* | *NOMAD* | *NOMAD-dm* | *MATSuMoTo* |
|---|---|---|---|---|---|---|
| abhi n=3 K=4 | 40 (29) | 150 (9) | 161 (9) | 59 (20) | 129 (56) | 56 (26) |
| abhi n=3 K=10 | 42 (24) | 1001 (9) | 1000 (9) | 49 (20) | 122 (34) | 94 (64) |
| abhi n=3 K=20 | 41 (24) | 1001 (9) | 1001 (9) | 49 (20) | 122 (34) | 153 (123) |
| abhi n=4 K=4 | 75 (50) | 959 (12) | 993 (12) | 110 (16) | 453 (88) | 82 (52) |
| abhi n=4 K=10 | 89 (48) | 1001 (12) | 1001 (12) | 110 (16) | 480 (88) | 344 (314) |
| abhi n=4 K=20 | 101 (48) | 1001 (12) | 1001 (12) | 124 (30) | 488 (91) | 560 (530) |
| abhi n=5 K=4 | 154 (113) | 1000 (15) | 1001 (15) | 301 (41) | 1000 (156) | 133 (103) |
| lse n=3 K=4 | 20 (15) | 53 (13) | 70 (16) | 24 (6) | 43 (6) | 39 (9) |
| lse n=3 K=10 | 19 (19) | 414 (19) | 448 (21) | 32 (7) | 49 (7) | 40 (10) |
| lse n=3 K=20 | 20 (20) | 1000 (22) | 1000 (24) | 41 (7) | 50 (7) | 40 (10) |
| lse n=4 K=4 | 75 (12) | 181 (17) | 223 (22) | 61 (27) | 115 (15) | 42 (12) |
| lse n=4 K=10 | 108 (14) | 974 (25) | 1000 (29) | 82 (33) | 133 (25) | 42 (12) |
| lse n=4 K=20 | 708 (16) | 983 (29) | 1000 (33) | 64 (9) | 129 (9) | 47 (17) |
| lse n=5 K=4 | 199 (14) | 760 (21) | 826 (51) | 67 (8) | 224 (8) | 44 (14) |
| CB3I n=3 K=4 | 25 (10) | 181 (6) | 184 (6) | 56 (12) | 108 (12) | 64 (34) |
| CB3I n=3 K=10 | 26 (10) | 1001 (6) | 1000 (6) | 56 (12) | 108 (12) | 226 (196) |
| CB3I n=3 K=20 | 24 (10) | 1001 (6) | 1000 (6) | 56 (12) | 108 (12) | 1000 (1000) |
| CB3I n=4 K=4 | 54 (14) | 1001 (8) | 1001 (57) | 149 (42) | 456 (23) | 96 (66) |
| CB3I n=4 K=10 | 59 (14) | 1001 (8) | 1001 (57) | 150 (48) | 457 (23) | 1000 (1000) |
| CB3I n=4 K=20 | 64 (14) | 1001 (8) | 1001 (57) | 154 (35) | 471 (35) | 1000 (1000) |
| CB3I n=5 K=4 | 155 (68) | 1001 (10) | 1000 (89) | 417 (18) | 1000 (18) | 237 (207) |
| CB3II n=3 K=4 | 26 (11) | 181 (6) | 184 (6) | 44 (12) | 108 (12) | 59 (29) |
| CB3II n=3 K=10 | 30 (11) | 1001 (6) | 1000 (6) | 44 (12) | 108 (12) | 227 (197) |
| CB3II n=3 K=20 | 34 (11) | 1001 (6) | 1000 (6) | 44 (12) | 108 (12) | 1000 (1000) |
| CB3II n=4 K=4 | 56 (14) | 1001 (18) | 1001 (49) | 125 (30) | 460 (35) | 101 (71) |
| CB3II n=4 K=10 | 54 (14) | 1001 (18) | 1001 (49) | 125 (30) | 462 (35) | 1000 (1000) |
| CB3II n=4 K=20 | 53 (14) | 1001 (18) | 1001 (49) | 173 (61) | 477 (51) | 1000 (1000) |
| CB3II n=5 K=4 | 135 (66) | 1001 (25) | 1001 (85) | 465 (69) | 1000 (54) | 197 (167) |
| LQ n=3 K=4 | 26 (2) | 181 (4) | 182 (4) | 48 (7) | 107 (7) | 44 (14) |
| LQ n=3 K=10 | 34 (2) | 1001 (4) | 1000 (4) | 48 (7) | 107 (7) | 49 (19) |
| LQ n=3 K=20 | 47 (2) | 1001 (4) | 1000 (4) | 48 (7) | 107 (7) | 54 (24) |
| LQ n=4 K=4 | 52 (15) | 1001 (6) | 1000 (6) | 145 (9) | 453 (10) | 46 (16) |
| LQ n=4 K=10 | 51 (15) | 1001 (6) | 1000 (6) | 145 (9) | 454 (10) | 86 (56) |
| LQ n=4 K=20 | 49 (15) | 1001 (6) | 1000 (6) | 157 (20) | 471 (20) | 81 (51) |
| LQ n=5 K=4 | 129 (17) | 1001 (8) | 1001 (8) | 425 (15) | 1000 (15) | 54 (24) |
| entropy n=3 K=4 | 28 (1) | 181 (1) | 181 (1) | 36 (1) | 108 (1) | 44 (14) |
| entropy n=3 K=10 | 21 (1) | 1001 (1) | 1001 (1) | 36 (1) | 108 (1) | 60 (30) |
| entropy n=3 K=20 | 36 (1) | 1001 (1) | 1001 (1) | 36 (1) | 108 (1) | 54 (24) |
| entropy n=4 K=4 | 51 (1) | 1001 (1) | 1001 (1) | 89 (1) | 451 (1) | 56 (26) |
| entropy n=4 K=10 | 50 (1) | 1001 (1) | 1001 (1) | 89 (1) | 451 (1) | 94 (64) |
| entropy n=4 K=20 | 50 (1) | 1001 (1) | 1001 (1) | 110 (1) | 465 (1) | 417 (387) |
| entropy n=5 K=4 | 143 (1) | 1001 (1) | 1001 (1) | 254 (1) | 1000 (1) | 63 (33) |
| infnorm n=3 K=4 | 18 (1) | 181 (1) | 181 (1) | 34 (1) | 107 (1) | 49 (19) |
| infnorm n=3 K=10 | 19 (1) | 1001 (1) | 1001 (1) | 34 (1) | 107 (1) | 59 (29) |
| infnorm n=3 K=20 | 19 (1) | 1001 (1) | 1001 (1) | 34 (1) | 107 (1) | 60 (30) |
| infnorm n=4 K=4 | 35 (1) | 1001 (1) | 1001 (1) | 90 (1) | 451 (1) | 57 (27) |
| infnorm n=4 K=10 | 34 (1) | 1001 (1) | 1001 (1) | 90 (1) | 451 (1) | 277 (247) |
| infnorm n=4 K=20 | 34 (1) | 1001 (1) | 1001 (1) | 116 (1) | 465 (1) | 261 (231) |
| infnorm n=5 K=4 | 82 (1) | 1001 (1) | 1001 (1) | 255 (1) | 1000 (1) | 107 (77) |

**Table A.2:** Number of function evaluations taken by solvers for 7 of the 16 base problems and various values of $n$ and $K$. For *MATSuMoTo*, the nearest value less than the median of the number of the evaluations of 20 replications is chosen.

| Problem | *SUCIL* | *DFLINT* | *DFLINT-M* | *NOMAD* | *NOMAD-dm* | *MATSuMoTo* |
|---|---|---|---|---|---|---|
| KLT n=3 K=4 | 27 (10) | 146 (10) | 152 (24) | 51 (24) | 116 (34) | 49 (19) |
| KLT n=3 K=10 | 33 (20) | 1001 (10) | 1001 (27) | 52 (25) | 125 (55) | 55 (25) |
| KLT n=3 K=20 | 33 (20) | 1001 (10) | 1001 (27) | 52 (25) | 125 (55) | 147 (117) |
| KLT n=4 K=4 | 75 (40) | 953 (13) | 954 (125) | 116 (42) | 457 (55) | 56 (26) |
| KLT n=4 K=10 | 68 (33) | 1001 (13) | 1000 (132) | 121 (30) | 492 (71) | 568 (538) |
| KLT n=4 K=20 | 71 (33) | 1001 (13) | 1000 (132) | 112 (30) | 469 (30) | 56 (26) |
| KLT n=5 K=4 | 121 (69) | 1001 (16) | 1000 (169) | 296 (43) | 1000 (176) | 90 (60) |
| logfrac n=3 K=4 | 18 (17) | 53 (13) | 70 (16) | 24 (6) | 43 (6) | 39 (9) |
| logfrac n=3 K=10 | 22 (12) | 414 (19) | 448 (21) | 31 (7) | 49 (7) | 39 (9) |
| logfrac n=3 K=20 | 47 (39) | 1000 (22) | 1000 (24) | 48 (7) | 50 (7) | 39 (9) |
| logfrac n=4 K=4 | 28 (12) | 181 (17) | 223 (22) | 51 (17) | 116 (13) | 41 (11) |
| logfrac n=4 K=10 | 29 (14) | 974 (25) | 1000 (29) | 95 (38) | 126 (18) | 41 (11) |
| logfrac n=4 K=20 | 31 (16) | 983 (29) | 1000 (33) | 150 (9) | 157 (9) | 41 (11) |
| logfrac n=5 K=4 | 46 (14) | 760 (21) | 810 (51) | 59 (8) | 224 (8) | 48 (18) |
| maxq n=3 K=4 | 18 (1) | 181 (1) | 181 (1) | 34 (1) | 107 (1) | 49 (19) |
| maxq n=3 K=10 | 19 (1) | 1001 (1) | 1001 (1) | 34 (1) | 107 (1) | 69 (39) |
| maxq n=3 K=20 | 19 (1) | 1001 (1) | 1001 (1) | 34 (1) | 107 (1) | 108 (78) |
| maxq n=4 K=4 | 35 (1) | 1001 (1) | 1001 (1) | 91 (1) | 451 (1) | 62 (32) |
| maxq n=4 K=10 | 34 (1) | 1001 (1) | 1001 (1) | 91 (1) | 451 (1) | 226 (196) |
| maxq n=4 K=20 | 34 (1) | 1001 (1) | 1001 (1) | 118 (1) | 465 (1) | 81 (51) |
| maxq n=5 K=4 | 82 (1) | 1001 (1) | 1001 (1) | 257 (1) | 1000 (1) | 117 (87) |
| multlin n=3 K=4 | 20 (20) | 64 (28) | 77 (28) | 32 (13) | 92 (54) | 40 (10) |
| multlin n=3 K=10 | 24 (24) | 419 (30) | 459 (30) | 34 (14) | 95 (55) | 66 (36) |
| multlin n=3 K=20 | 26 (26) | 1000 (31) | 1000 (31) | 47 (15) | 105 (56) | 109 (79) |
| multlin n=4 K=4 | 38 (38) | 239 (81) | 257 (81) | 96 (44) | 256 (91) | 51 (21) |
| multlin n=4 K=10 | 60 (60) | 1000 (83) | 1000 (83) | 107 (45) | 266 (92) | 128 (98) |
| multlin n=4 K=20 | 76 (76) | 1000 (84) | 1000 (84) | 108 (38) | 272 (96) | 231 (201) |
| multlin n=5 K=4 | 38 (37) | 923 (192) | 949 (192) | 224 (40) | 1000 (719) | 111 (81) |
| mxhilb n=3 K=4 | 19 (1) | 181 (1) | 181 (1) | 35 (1) | 106 (1) | 50 (20) |
| mxhilb n=3 K=10 | 19 (1) | 1001 (1) | 1001 (1) | 35 (1) | 106 (1) | 69 (39) |
| mxhilb n=3 K=20 | 19 (1) | 1001 (1) | 1001 (1) | 35 (1) | 106 (1) | 232 (202) |
| mxhilb n=4 K=4 | 56 (1) | 1001 (1) | 1001 (1) | 90 (1) | 450 (1) | 67 (37) |
| mxhilb n=4 K=10 | 58 (1) | 1001 (1) | 1001 (1) | 90 (1) | 450 (1) | 407 (377) |
| mxhilb n=4 K=20 | 58 (1) | 1001 (1) | 1001 (1) | 113 (1) | 464 (1) | 1000 (1000) |
| mxhilb n=5 K=4 | 144 (1) | 1001 (1) | 1001 (1) | 257 (1) | 1000 (1) | 88 (58) |
| onenorm n=3 K=4 | 19 (1) | 181 (1) | 181 (1) | 36 (1) | 107 (1) | 44 (14) |
| onenorm n=3 K=10 | 19 (1) | 1001 (1) | 1001 (1) | 36 (1) | 107 (1) | 60 (30) |
| onenorm n=3 K=20 | 19 (1) | 1001 (1) | 1001 (1) | 36 (1) | 107 (1) | 159 (129) |
| onenorm n=4 K=4 | 48 (1) | 1001 (1) | 1001 (1) | 89 (1) | 451 (1) | 57 (27) |
| onenorm n=4 K=10 | 48 (1) | 1001 (1) | 1001 (1) | 89 (1) | 451 (1) | 240 (210) |
| onenorm n=4 K=20 | 48 (1) | 1001 (1) | 1001 (1) | 117 (1) | 469 (1) | 1000 (1000) |
| onenorm n=5 K=4 | 133 (1) | 1001 (1) | 1001 (1) | 255 (1) | 1000 (1) | 83 (53) |
| quad n=3 K=4 | 27 (10) | 150 (9) | 157 (51) | 53 (18) | 119 (35) | 44 (14) |
| quad n=3 K=10 | 34 (11) | 1001 (9) | 1000 (60) | 53 (22) | 125 (49) | 59 (29) |
| quad n=3 K=20 | 30 (11) | 1001 (9) | 1000 (60) | 53 (22) | 125 (49) | 113 (83) |
| quad n=4 K=4 | 103 (47) | 959 (12) | 982 (113) | 111 (12) | 460 (89) | 56 (26) |
| quad n=4 K=10 | 66 (37) | 1001 (12) | 1000 (128) | 112 (20) | 486 (73) | 224 (194) |
| quad n=4 K=20 | 66 (37) | 1001 (12) | 1000 (128) | 115 (30) | 472 (57) | 102 (72) |
| quad n=5 K=4 | 146 (58) | 1000 (15) | 1001 (171) | 286 (26) | 1000 (58) | 116 (86) |

**Table A.3:** Number of function evaluations taken by solvers for 2 of the 16 base problems and various values of $n$ and $K$. For *MATSuMoTo*, the nearest value less than the median of the number of the evaluations of 20 replications is chosen.

| Problem | *SUCIL* | *DFLINT* | *DFLINT-M* | *NOMAD* | *NOMAD-dm* | *MATSuMoTo* |
|---|---|---|---|---|---|---|
| recipro n=3 K=4 | 17 (16) | 50 (10) | 61 (10) | 32 (7) | 47 (8) | 39 (9) |
| recipro n=3 K=10 | 21 (21) | 411 (16) | 444 (16) | 50 (16) | 64 (15) | 44 (14) |
| recipro n=3 K=20 | 22 (14) | 1000 (19) | 1000 (19) | 43 (8) | 64 (9) | 44 (14) |
| recipro n=4 K=4 | 24 (24) | 177 (13) | 218 (13) | 75 (21) | 198 (41) | 41 (11) |
| recipro n=4 K=10 | 27 (14) | 970 (21) | 1000 (21) | 111 (40) | 216 (30) | 51 (21) |
| recipro n=4 K=20 | 29 (16) | 979 (25) | 1000 (25) | 128 (60) | 243 (59) | 61 (31) |
| recipro n=5 K=4 | 44 (14) | 755 (16) | 814 (16) | 224 (47) | 653 (40) | 53 (23) |
| reciprob n=3 K=4 | 18 (16) | 50 (10) | 61 (10) | 32 (7) | 47 (8) | 39 (9) |
| reciprob n=3 K=10 | 22 (21) | 411 (16) | 438 (16) | 53 (15) | 64 (11) | 44 (14) |
| reciprob n=3 K=20 | 32 (31) | 1000 (19) | 1000 (19) | 61 (12) | 64 (9) | 49 (19) |
| reciprob n=4 K=4 | 26 (12) | 177 (13) | 199 (13) | 80 (21) | 207 (28) | 42 (12) |
| reciprob n=4 K=10 | 39 (39) | 970 (21) | 1000 (21) | 111 (31) | 216 (30) | 46 (16) |
| reciprob n=4 K=20 | 64 (38) | 979 (25) | 1000 (25) | 82 (9) | 237 (73) | 61 (31) |
| reciprob n=5 K=4 | 44 (14) | 755 (16) | 788 (16) | 186 (26) | 685 (70) | 53 (23) |

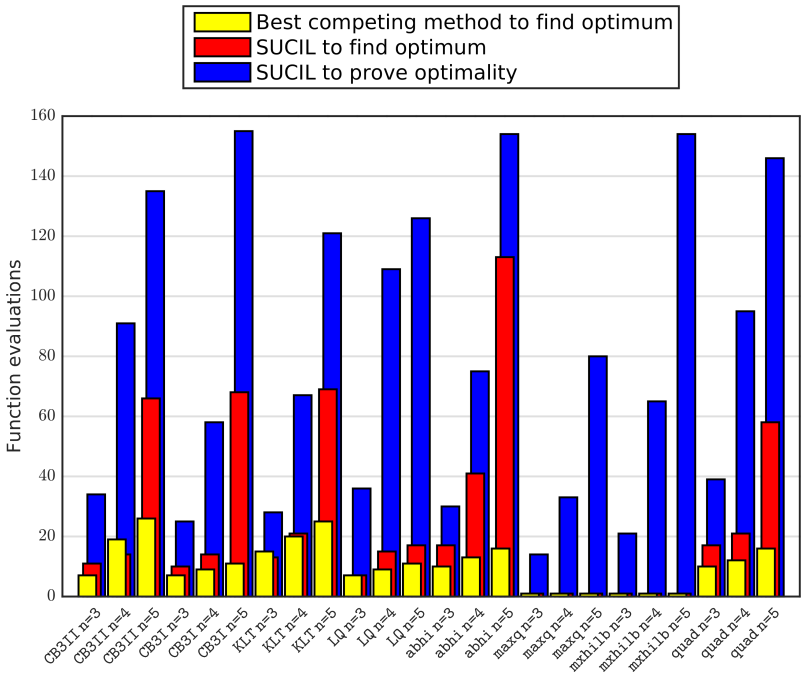



**Figure A.1:** Number of evaluations for 8 test functions on $[-4,4]^5 \cap \mathbb{Z}^n$ before *SUCIL* first identifies a global minimum and evaluations required to prove its global optimality. The fewest number of evaluations required by any of *DFLINT*, *DFLINT-M*, *NOMAD*, *NOMAD-dm*, and *MATSuMoTo* is shown for comparison.

**Table A.4:** Set of convex test problems and their minimizers on the domain $[-K, K]^n \cap \mathbb{Z}^n$. The corresponding references are: [1] **?** [2] **?** [3] **?** [4] **?** [5] **?** [6] **?**, Section 6.4 [7] **?** [8] **?**, Exercise 2.2.12 [9] **?** [10] **?**, Example 2.2.9 [11] **?** [12] **?**, Exercise 4.4.10 [13] **?**, Exercise 4.4.10.

| Name | Expression | $f(x^*)$ | $x^*$ |
|---|---|---|---|
| `abhi` [1] | $\sum_{i=1}^{n}\left[64\left(c_1(x_i-2)-c_2(x_{i+1}-2)\right)^2+\left(c_2(x_i-2)-c_1(x_{i+1}-2)\right)^2\right]$, $c_1=\cos\left(\frac{\pi}{8}\right), c_2=\sin\left(\frac{\pi}{8}\right)$ | 0 | $2e$ |
| `lse` [2] | $\log\left(\sum_{i=1}^{n} e^{x_i}\right)$ | $\log(n)-K$ | $-Ke$ |
| `CB3I` [3] | $\sum_{i=1}^{n-1}\max\left\{x_i^4+x_{i+1}^2,(2-x_i)^2+(2-x_{i+1})^2,2e^{-x_i+x_{i+1}}\right\}$ | $2(n-1)$ | $e$ |
| `CB3II` [4] | $\max\left\{\sum_{i=1}^{n-1}x_i^4+x_{i+1}^2,\sum_{i=1}^{n-1}(2-x_i)^2+(2-x_{i+1})^2,\sum_{i=1}^{n-1}2e^{-x_i+x_{i+1}}\right\}$ | $2(n-1)$ | $e$ |
| `LQ` [5] | $\sum_{i=1}^{n-1}\max\left\{-x_i-x_{i+1},-x_i-x_{i+1}+x_i^2+x_{i+1}^2-1\right\}$ | $-(n-1)$ | many |
| `entropy` [6] | $\sum_{i=1}^{n}\lvert x_i\rvert\log\left(\lvert x_i\rvert+(1+\lvert x_i\rvert^2)^{1/2}\right)-\left(1+\lvert x_i\rvert^2\right)^{1/2}$ | $-n$ | 0 |
| `infnorm` | $\max_{i\in\{1\ldots n\}}\{\lvert x_i\rvert\}$ | 0 | 0 |
| `KLT` [7] | $\max_{i\in\{1,\ldots,n\}}\left\{\lVert x-c_i-2e\rVert^2\right\}, c_i=2e_i-e$ | $n$ | $2e$ |
| `logfrac` [8] | $\sum_{i:x_i\neq 0}\log\left(1+\frac{e^{2x_i}-1}{e^{x_i}-1}\right)+\sum_{i:x_i=0}\log(3), i\in\{1,\ldots,n\}$ | $n\log\left(1+\frac{e^{-2K}-1}{e^{-K}-1}\right)$ | $-Ke$ |
| `maxq` [9] | $\max_{i\in\{1,\ldots,n\}}\left\{x_i^2\right\}$ | 0 | 0 |
| `multlin` [10] | $-\left(\prod_{i=1}^{n}x_i\right)^{1/n}$ if $x_i\geq 0\ \forall i$, otherwise $\infty$ | $-K$ | $Ke$ |
| `mxhilb` [11] | $\max_{i\in\{1,\ldots,n\}}\left\{\sum_{j=1}^{n}\left\lvert\frac{x_j}{i+j-1}\right\rvert\right\}$ | 0 | 0 |
| `onenorm` | $\sum_{i=1}^{n}\lvert x_i\rvert$ | 0 | 0 |
| `quad` | $\sum_{i=1}^{n}(x_i-2)^2$ | 0 | $2e$ |
| `recipro` [12] | $\sum_{\mathcal{S}_j\subseteq\{1,\ldots,n\},\lvert\mathcal{S}_j\rvert\geq 1}\frac{(-1)^{\lvert\mathcal{S}_j+1\rvert}}{\log\left(2+\lvert\mathcal{S}_j\rvert K+\sum_{i\in\mathcal{S}_j}x_i\right)}$ | $\sum_{\mathcal{S}_j}\frac{(-1)^{\lvert\mathcal{S}_j+1\rvert}}{\log\left(2+\lvert\mathcal{S}_j\rvert(K+1)\right)}$ | $Ke$ |
| `reciprob` [13] | $\sum_{\mathcal{S}_j\subseteq\{1,\ldots,n\},\lvert\mathcal{S}_j\rvert\geq 1}\frac{(-1)^{\lvert\mathcal{S}_j+1\rvert}}{\lvert\mathcal{S}_j\rvert(K+1)+\sum_{i\in\mathcal{S}_j}x_i}$ | $\sum_{\mathcal{S}_j}\frac{(-1)^{\lvert\mathcal{S}_j+1\rvert}}{\lvert\mathcal{S}_j\rvert(2K+1)}$ | $Ke$ |

# Bibliography


Abhishek, K., Leyffer, S., and Linderoth, J., 2010a, "FilMINT: An outer approximation based solver for convex mixed-integer nonlinear programs," *Informs Journal on Computing* **22**, 555–567.

Abhishek, K., Leyffer, S., and Linderoth, J. T., 2010b, "Modeling without categorical variables: A mixed-integer nonlinear program for the optimization of thermal insulation systems," *Optimization and Engineering* **11**, 185–212.

Abramson, M., Audet, C., Chrissis, J., and Walston, J., 2009, "Mesh adaptive direct search algorithms for mixed variable optimization," *Optimization Letters* **3**, 35–47.

Achterberg, T., 2007a, "Conflict analysis in mixed integer programming," *Discrete Optimization* **4**, 4–20.

Achterberg, T., 2007b, *Constraint integer programming*, Ph.D. thesis (Technical University Berlin).

Achterberg, T., 2009, "SCIP: solving constraint integer programs," *Mathematical Programming Computation* **1**, 1–41.

Achterberg, T., Bixby, R. E., Gu, Z., Rothberg, E., and Weninger, D., 2016, *Presolve Reductions in Mixed Integer Programming*, Tech. Rep. 16-44 (ZIB, Takustr. 7, 14195 Berlin).

Achterberg, T., Koch, T., and Martin, A., 2005, "Branching rules revisited," *Operations Research Letters* **33**, 42–54.

Androulakis, I. P., Maranas, C. D., and Floudas, C. A., 1995, "$\alpha$BB: A global optimization method for general constrained nonconvex problems," *Journal of Global Optimization* **7**, 337–363.

Applegate, D., Bixby, R., Chvátal, V., and Cook, W., 1998, "On the solution of traveling salesman problems," in *Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung, International Congress of Mathematicians*, pp. 645–656.

Audet, C., and Dennis, Jr., J. E., 2000, "Pattern search algorithms for mixed variable programming," *SIAM Journal on Optimization* **11**, 573–594.

Audet, C., and Dennis Jr, J. E., 2006, "Mesh adaptive direct search algorithms for constrained optimization," *SIAM Journal on optimization* **17**, 188–217.

Audet, C., and Hare, W. L., 2017, *Derivative-Free and Blackbox Optimization* (Springer).

Audet, C., Le Digabel, S., and Tribes, C., 2019, "The mesh adaptive direct search algorithm for granular and discrete variables," *SIAM Journal on Optimization* **29**, 1164–1189.

Balaprakash, P., Tiwari, A., and Wild, S. M., 2014, "Multi-objective optimization of HPC kernels for performance, power, and energy," in *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, Vol. 8551, edited by Jarvis, S. A., Wright, S. A., and Hammond, S. D. (Springer). pp. 239–260.

Bartz-Beielstein, T., and Zaefferer, M., 2017, "Model-based methods for continuous and discrete global optimization," *Applied Soft Computing* **55**, 154–167.

Belotti, P., Lee, J., Liberti, L., Margot, F., and Wachter, A., 2009, "Branching and bounds tightening techniques for non-convex MINLP," *Optimization Methods and Software* **24**

Belotti, P., 2009, *Couenne: a user's manual*, Tech. Rep. (Technical report, Lehigh University).

Belotti, P., Kirches, C., Leyffer, S., Linderoth, J., Luedtke, J., and Mahajan, A., 2013, "Mixed-integer nonlinear optimization," *Acta Numerica* **22**, 1–131.

Berthold, T., 2006, *Primal Heuristics for Mixed Integer Programs*, Master's thesis

Berthold, T., 2014a, *Heuristic algorithms in global MINLP solvers*, Ph.D. thesis

Berthold, T., 2014b, "Primal MINLP heuristics in a nutshell," in *Operations Research Proceedings 2013* (Springer). pp. 23–28.

Berthold, T., 2018, "A computational study of primal heuristics inside an MI(NL)P solver," *Journal of Global Optimization* **70**, 189–206.

Berthold, T., Farmer, J., Heinz, S., and Perregaard, M., 2018, "Parallelization of the FICO Xpress-Optimizer," *Optimization Methods and Software* **33**, 518–529.

Berthold, T., and Gleixner, A. M., 2014, "Undercover: a primal MINLP heuristic exploring a largest sub-MIP," *Mathematical Programming* **144**, 315–346.

Bixby, R., and Rothberg, E., 2007, "Progress in computational mixed integer programming–a look back from the other side of the tipping point," *Annals of Operations Research* **149**, 37.

Bonami, P., Biegler, L., Conn, A., Cornuéjols, G., Grossmann, I., Laird, C., Lee, J., Lodi, A., Margot, F., Sawaya, N., and Wächter, A., 2008a, "An algorithmic framework for convex mixed integer nonlinear programs," *Discrete Optimization* **5**, 186–204.

Bonami, P., Biegler, L. T., Conn, A. R., Cornuéjols, G., Grossmann, I. E., Laird, C. D., Lee, J., Lodi, A., Margot, F., Sawaya, N., and Wächter, A., 2008b, "An algorithmic framework for convex mixed integer nonlinear programs," *Discrete Optimization* **5**, 186–204.

Bonami, P., Cornuéjols, G., Lodi, A., and Margot, F., 2009, "A feasibility pump for mixed integer nonlinear programs," *Mathematical Programming* **119**, 331–352.

Bonami, P., and Gonçalves, J. P., 2012, "Heuristics for convex mixed integer nonlinear programs," *Computational Optimization and Applications* **51**, 729–747.

Bonami, P., and Lee, J., 2007, "BONMIN user's manual," *Numer. Math.* **4**, 1–32.

Borwein, J. M., Bailey, D. H., Girgensohn, R., Bailey, D. H., and Borwein, J. M., 2004, *Experimentation in Mathematics: Computational Paths to Discovery* (CRC Press).

Borwein, J. M., and Vanderwerff, J. D., 2010, *Convex Functions: Constructions, Characterizations and Counterexamples* (Cambridge University Press).

Boukouvala, F., Misener, R., and Floudas, C. A., 2016a, "Global optimization advances in mixed-integer nonlinear programming, MINLP, and constrained derivative-free optimization, CDFO," *European Journal of Operational Research* **252**, 701–727.

Boukouvala, F., Misener, R., and Floudas, C. A., 2016b, "Global optimization advances in mixed-integer nonlinear programming, MINLP, and constrained derivative-free optimization, CDFO," *European Journal of Operational Research* **252**, 701–727.

Boyd, S., and Vandenberghe, L., 2004, *Convex Optimization* (Cambridge University Press).

Brooke, A., Kendrick, D., Meeraus, A., and Raman, R., 1998, *GAMS A user's guide*, GAMS Developments Corporation, 1217 Potomac Street, N.W., Washington DC 20007, USA

Buchheim, C., Kuhlmann, R., and Meyer, C., 2018, "Combinatorial optimal control of semilinear elliptic PDEs," *Computational Optimization and Applications* **70**, 641–675.

Burer, S., and Letchford, A. N., 2012, "Non-convex mixed-integer nonlinear programming: A survey," *Surveys in Operations Research and Management Science* **17**, 97–106.

Bussieck, M. R., Drud, A. S., and Meeraus, A., 2003, "MINLPLib - a collection of test models for mixed-integer nonlinear programming," *INFORMS Journal on Computing* **15**, 114–119.

CBC. "CBC User Guide," , 2019. URL `http://www.coin-or.org/Cbc`

Chapman, B., Jost, G., and Van Der Pas, R., 2008, *Using OpenMP: portable shared memory parallel programming*, Vol. 10 (MIT press).

Charalambous, C., and Bandler, J. W., 1976, "Non-linear minimax optimization as a sequence of least $p$th optimization with finite values of $p$," *International Journal of Systems Science* **7**, 377–391.

Charalambous, C., and Conn, A. R., 1978, "An efficient method to solve the minimax problem directly," *SIAM Journal on Numerical Analysis* **15**, 162–187.

Chinneck, J., and Shafique, M., 2014, "Towards a fast heuristic for MINLP," presented at Mixed-Integer Nonlinear Programming 2014, CMU, Pittsburgh.

Conn, A. R., Scheinberg, K., and Vicente, L. N., 2009, *Introduction to Derivative-Free Optimization* (SIAM).

Costa, A., and Nannicini, G., 2018, "RBFOpt: An open-source library for black-box optimization with costly function evaluations," *Mathematical Programming Computation* **10**, 597–629.

CPLEX. "CPLEX 12.8 User's Manual," , 2019. URL https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.studio.help/pdf/usrcplex.pdf

Crainic, T. G., Le Cun, B., and Roucairol, C., 2006, "Parallel branch-and-bound algorithms," *Parallel combinatorial optimization*, 1–28.

Dagum, L., and Menon, R., 1998, "OpenMP: an industry standard API for shared-memory programming," *IEEE computational science and engineering* **5**, 46–55.

D'Ambrosio, C., Frangioni, A., Liberti, L., and Lodi, A., 2012, "A storm of feasibility pumps for nonconvex MINLP," *Mathematical programming* **136**, 375–402.

Danna, E., Rothberg, E., and Le Pape, C., 2005, "Exploring relaxation induced neighborhoods to improve MIP solutions," *Mathematical Programming* **102**, 71–90.

Davis, E., and Ierapetritou, M., 2009, "A kriging based method for the solution of mixed-integer nonlinear programs containing black-box functions," *Journal of Global Optimization* **43**, 191–205.

Dennard, R. H., Gaensslen, F. H., Yu, H.-N., Rideout, V. L., Bassous, E., and LeBlanc, A. R., 1974, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits* **9**, 256–268.

Dennis, Jr., J. E., and Woods, D. J., 1987, "Optimization on microcomputers: The Nelder-Mead simplex algorithm," in *New Computing Environments: Microcomputers in Large-Scale Computing*, edited by Wouk, A. (SIAM). ISBN 0898712106, pp. 116–122.

Dolan, E. D., and Moré, J., 2002, "Benchmarking optimization software with performance profiles," *Mathematical Programming*, A **91**, 201–213.

Duran, M. A., and Grossmann, I. E., 1986, "An outer-approximation algorithm for a class of mixed-integer nonlinear programs," *Mathematical Programming* **36**, 307–339.

Fletcher, R., 2000, *Practical Methods of Optimization*, 2nd ed. (John Wiley & Sons, Chichester).

Fletcher, R., and Leyffer, S., 1994, "Solving mixed integer nonlinear programs by outer approximation," *Mathematical Programming* **66**, 327–349.

Floudas, C. A., and Gounaris, C. E., 2009, "A review of recent advances in global optimization," *Journal of Global Optimization* **45**, 3.

Fourer, R., Gay, D. M., and Kernighan, B. W., 2003, *AMPL: A Modelling Language for Mathematical Programming*, 2nd ed. (Books/Cole—Thomson Learning).

Fujie, T., and Kojima, M., 1997, "Semidefinite programming relaxation for nonconvex quadratic programs," *Journal of Global Optimization* **10**, 367–380.

García-Palomares, U. M., and Rodríguez-Hernández, P. S., 2019, "Unified approach for solving box-constrained models with continuous or discrete variables by non monotone direct search methods," *Optimization Letters* **13**, 95–111.

Geißler, B., Martin, A., Morsi, A., and Schewe, L., 2012, "Using piecewise linear functions for solving MINLPs," in *Mixed integer nonlinear programming* (Springer). pp. 287–314.

Geoffrion, A. M., and Marsten, R. E., 1972, "Integer programming algorithms: A framework and state-of-the-art survey," *Management Science* **18**, 465–491.

Geoffrion, A. M., 1972, "Generalized benders decomposition," *Journal of optimization theory and applications* **10**, 237–260.

Goux, J.-P., and Leyffer, S., 2002, "Solving large MINLPs on computational grids," *Optimization and Engineering* **3**, 327–346.

Graf, P. A., and Billups, S., 2017, "MDTri: Robust and efficient global mixed integer search of spaces of multiple ternary alloys," *Computational Optimization and Applications* **68**, 671–687.

Grama, A., Kumar, V., Gupta, A., and Karypis, G., 2003, *Introduction to parallel computing* (Pearson Education).

Gümüş, Z. H., and Floudas, C. A., 2005, "Global optimization of mixed-integer bilevel programming problems," *Computational Management Science* **2**, 181–212.

Gupta, O. K., and Ravindran, A., 1985, "Branch and bound experiments in convex nonlinear integer programming," *Management science* **31**, 1533–1546.

GUROBI. "Gurobi optimizer 9.0 reference manual," , 2019. URL https://www.gurobi.com/wp-content/plugins/hd_documentations/documentation/9.0/refman.pdf

Gutmann, H.-M., 2001, "A radial basis function method for global optimization," *Journal of global optimization* **19**, 201–227.

Haarala, M., Miettinen, K., and Mäkelä, M. M., 2004, "New limited memory bundle method for large-scale nonsmooth optimization," *Optimization Methods and Software* **19**, 673–692.

Hall, J., 2010, "Towards a practical parallelisation of the simplex method," *Computational Management Science* **7**, 139–170.

Hart, W. E., Watson, J.-P., and Woodruff, D. L., 2011, "Pyomo: modeling and solving mathematical programs in Python," *Mathematical Programming Computations* **3**, 219–260.

Hart, W. E., Phillips, C. A., and Eckstein, J., 2013, *PEBBL: An object-oriented framework for scalable parallel branch and bound.*, Tech. Rep. (Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States)).

Hart, W. E., and Phillips, C. A., 2008, *Parallelization Issues for MINLP.*, Tech. Rep. (Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States)).

Hemker, T., Fowler, K., Farthing, M., and von Stryk, O., 2008, "A mixed-integer simulation-based optimization approach with surrogate functions in water resources management," *Optimization and Engineering* **9**, 341–360.

Hemmecke, R., Köppe, M., Lee, J., and Weismantel, R., 2010, "Nonlinear integer programming," in *50 Years of Integer Programming 1958–2008* (Springer). pp. 561–618.

Holmström, K., Quttineh, N.-H., and Edvall, M., 2008, "An adaptive radial basis algorithm (ARBF) for expensive black-box mixed-integer constrained global optimization," *Optimization and Engineering* **9**, 311–339.

Hunting, M., 2011, "The AIMMS outer approximation algorithm for MINLP," *Paragon Decision Technology, Haarlem*

Hunting, M. "AIMMS– Whitepaper-COA," , 2019. URL https://download.aimms.com/aimms/download/references/AIMMS-Whitepaper-COA.pdf

Jeroslow, R. C., 1973, "There cannot be any algorithm for integer programming with quadratic constraints," *Operations Research* **21**, 221–224.

Jian, N., and Henderson, S. G., 2020, "Estimating the probability that a function observed with noise is convex," *INFORMS Journal on Computing* **32**, 376–389.

Jian, N., Henderson, S. G., and Hunter, S. R., 2014, "Sequential detection of convexity from noisy function evaluations," in *Proceedings of the Winter Simulation Conference* (IEEE).

Kannan, R., and Monma, C. L., 1978, "On the computational complexity of integer programming problems," in *Optimization and Operations Research* (Springer). pp. 161–172.

Karmarkar, N., 1984, "A new polynomial-time algorithm for linear programming," in *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pp. 302–311.

Kilinç, M., and Sahinidis, N. V., 2017, "State-of-the-art in mixed-integer nonlinear programming," in *Advances and trends in optimization with engineering applications*, MOS-SIAM book series on optimization (SIAM, Philadelphia). pp. 273–292.

Kiwiel, K. C., 1989, "An ellipsoid trust region bundle method for nonsmooth convex minimization," *SIAM Journal on Control and Optimization* **27**, 737–757.

Kleinert, T., Grimm, V., and Schmidt, M., 2019, *Outer Approximation for Global Optimization of Mixed-Integer Quadratic Bilevel Problems*, Tech. Rep. (Friedrich-Alexander-Universität Erlangen-Nürnberg).

Koch, T., Ralphs, T., and Shinano, Y., 2012, "Could we use a million cores to solve an integer program?." *Mathematical Methods of Operations Research* **76**, 67–93.

Kolda, T. G., Lewis, R. M., and Torczon, V. J., 2003, "Optimization by direct search: New perspectives on some classical and modern methods," *SIAM Review* **45**, 385–482.

Kronqvist, J., Lundell, A., and Westerlund, T., 2016, "The extended supporting hyperplane algorithm for convex mixed-integer nonlinear programming," *Journal of Global Optimization* **64**, 249–272.

Lai, T.-H., and Sahni, S., 1984, "Anomalies in parallel branch-and-bound algorithms," *Communications of the ACM* **27**, 594–602.

Larson, J., Menickelly, M., and Wild, S. M., 2019, "Derivative-free optimization methods," *Acta Numerica* **28**, 287–404.

Lasdon, L., and Plummer, J. C., 2008, "Multistart algorithms for seeking feasibility," *Computers & operations research* **35**, 1379–1393.

Le Digabel, S., 2011, "Algorithm 909: NOMAD: Nonlinear optimization with the MADS algorithm," *ACM Transactions on Mathematical Software* **37**, 1–15.

Le Digabel, S., and Wild, S. M., 2015, *A Taxonomy of Constraints in Black-Box Simulation-Based Optimization*, Preprint ANL/MCS-P5350-0515 (Argonne).

Li, G.-J., 1985, *Parallel processing of combinatorial search problems*, Ph.D. thesis (Purdue University).

Li, G.-J., and Wah, B. W., 1986, "Coping with anomalies in parallel branch-and-bound algorithms," *IEEE Transactions on Computers* **100**, 568–573.

Li, G.-J., and Wah, B. W., 1990, "Computational efficiency of parallel combinatorial or-tree searches," *IEEE Transactions on Software Engineering* **16**, 13–31.

Lima, R. M., and Grossmann, I. E., 2011, "Computational advances in solving mixed integer linear programming problems," *Chemical Engineering Greetings to Prof. Sauro Pierucci, AIDAC* **151**, 160.

Lin, Y., and Schrage, L., 2009, "The global solver in the LINDO API," *Optimization Methods & Software* **24**, 657–668.

Linderoth, J. T., and Savelsbergh, M. W., 1999, "A computational study of search strategies for mixed integer programming," *INFORMS Journal on Computing* **11**, 173–187.

LINDO. "LINDO Systems Inc," , 2019. URL https://www.lindo.com/downloads/PDF/LindoUsersManual.pdf

Liuzzi, G., Lucidi, S., and Rinaldi, F., 2011, "Derivative-free methods for bound constrained mixed-integer optimization," *Computational Optimization and Applications* **53**, 505–526.

Liuzzi, G., Lucidi, S., and Rinaldi, F., 2015, "Derivative-free methods for mixed-integer constrained optimization problems," *Journal of Optimization Theory and Applications* **164**, 933–965.

Liuzzi, G., Lucidi, S., and Rinaldi, F., 2020, "An algorithmic framework based on primitive directions and nonmonotone line searches for black-box optimization problems with integer variables," *Mathematical Programming Computation* **12**, 673–702.

Lourenço, H. R., Martin, O. C., and Stützle, T., 2003, "Iterated local search," in *Handbook of metaheuristics* (Springer). pp. 320–353.

Lubin, M., Hall, J. J., Petra, C. G., and Anitescu, M., 2013, "Parallel distributed-memory simplex for large-scale stochastic LP problems," *Computational Optimization and Applications* **55**, 571–596.

Lundell, A., Kronqvist, J., and Westerlund, T., 2018, "The supporting hyperplane optimization toolkit-a polyhedral outer approximation based convex minlp solver utilizing a single branching tree approach.." *Preprint, Optimization Online*

Mahajan, A., 2010, "Presolving mixed–integer linear programs," *Wiley Encyclopedia of Operations Research and Management Science*

Mahajan, A., Leyffer, S., Linderoth, J., Luedtke, J., and Munson, T., 2020, "Minotaur: A mixed-integer nonlinear optimization toolkit," *Mathematical Programming Computation*, 1–38.

Mans, B., and Roucairol, C., 1996, "Performances of parallel branch and bound algorithms with best-first search," *Discrete Applied Mathematics* **66**, 57–74.

MATLAB. "Optimization Toolbox," , 2021. URL `https://www.mathworks.com/products/optimization.html`

McCormick, G. P., 1976, "Computability of global solutions to factorable nonconvex programs: Part I-Convex underestimating problems," *Mathematical programming* **10**, 147–175.

Melo, W., Fampa, M., and Raupp, F., 2018, "An overview of MINLP algorithms and their implementation in Muriqui Optimizer," *Annals of Operations Research*, 1–25.

Migdalas, A., Toraldo, G., and Kumar, V., 2003, "Nonlinear optimization and parallel computing," *Parallel Computing* **29**, 375–391.

Misener, R., and Floudas, C. A., 2014, "ANTIGONE: algorithms for continuous/integer global optimization of nonlinear equations," *Journal of Global Optimization* **59**, 503–526.

Mladenović, N., and Hansen, P., 1997, "Variable neighborhood search," *Computers & Operations Research* **24**, 1097–1100.

Moré, J. J., and Wild, S. M., 2009, "Benchmarking derivative-free optimization algorithms," *SIAM Journal on Optimization* **20**, 172–191.

Müller, J., 2014, *MATSuMoTo: The MATLAB surrogate model toolbox for computationally expensive black-box global optimization problems*, Tech. Rep. 1404.4261 (arXiv).

Müller, J., 2016, "MISO: Mixed-Integer Surrogate Optimization framework," *Optimization and Engineering* **17**, 177–203.

Müller, J., Shoemaker, C. A., and Piché, R., 2013, "SO-I: A surrogate model algorithm for expensive nonlinear integer programming problems including global optimization applications," *Journal of Global Optimization* **59**, 865–889.

Müller, J., Shoemaker, C. A., and Piché, R., 2013, "SO-MI: A surrogate model algorithm for computationally expensive nonlinear mixed-integer black-box global optimization problems," *Computers & Operations Research* **40**, 1383–1400.

Munguía, L.-M., Oxberry, G., and Rajan, D., 2016, "PIPS-SBB: A parallel distributed-memory branch-and-bound algorithm for stochastic mixed-integer programs," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (IEEE). pp. 730–739.

Munguía, L., Oxberry, G., Rajan, D., and Shinano, Y., 2019, "Parallel PIPS-SBB: multi-level parallelism for stochastic mixed-integer programs," *Comp. Opt. and Appl.* **73**, 575–601.

Nelder, J., and Mead, R., 1965, "A simplex method for function minimization," *Computer Journal* **7**, 308–313.

Nesterov, Y., and Nemirovskii, A., 1994, *Interior-point polynomial algorithms in convex programming* (SIAM).

Newby, E., and Ali, M. M., 2015 apr, "A trust-region-based derivative free algorithm for mixed integer programming," *Computational Optimization and Applications* **60**, 199–229.

Nocedal, J., and Wright, S. J., 1999, *Numerical Optimization* (Springer-Verlag, New York).

PARAM Siddhi-AI. "PARAM Siddhi-AI Supercomputer," , 2021. URL https://www.top500.org/system/179901/

Porcelli, M., and Toint, P. L., 2017 jun, "BFO, a trainable derivative-free brute force optimizer for nonlinear bound-constrained optimization and equilibrium computations with continuous and discrete variables," *ACM Transactions on Mathematical Software* **44**, 1–25.

Powell, M. J., 1999, "Recent research at cambridge on radial basis functions," *New Developments in Approximation Theory*, 215–232.

Quesada, I., and Grossmann, I. E., 1992, "An LP/NLP based branch and bound algorithm for convex MINLP optimization problems," *Computers & chemical engineering* **16**, 937–947.

Ralphs, T., Shinano, Y., Berthold, T., and Koch, T., 2018, "Parallel solvers for mixed integer linear optimization," in *Handbook of parallel constraint reasoning* (Springer). pp. 283–336.

Ralphs, T., Guzelsoy, M., and Mahajan, A., 2015, "SYMPHONY 5.6.9 user's manual,"

Rashid, K., Ambani, S., and Cetinkaya, E., 2012, "An adaptive multiquadric radial basis function method for expensive black-box mixed-integer nonlinear constrained optimization," *Engineering Optimization* **45**, 185–206.

Reeves, C. R., 2010, "Genetic algorithms," in *Handbook of metaheuristics* (Springer). pp. 109–139.

Richter, P., Ábrahám, E., and Morin, G., 2011, "Optimisation of concentrating solar thermal power plants with neural networks," in *Adaptive and Natural Com-*

*puting Algorithms*, Vol. 6593, edited by Dobnikar, A., Lotrič, U., and Šter, B. (Springer). pp. 190–199.

Rios, L. M., and Sahinidis, N. V., 2013, "Derivative-free optimization: a review of algorithms and comparison of software implementations," *Journal of Global Optimization* **56**, 1247–1293.

Rockafellar, R., 1970, *Convex Analysis* (Princeton University Press, Princeton, NJ).

Sacks, J., Welch, W. J., Mitchell, T. J., and Wynn, H. P., 1989, "Design and analysis of computer experiments," *Statistical science*, 409–423.

Sahinidis, N. V., 1996, "BARON: A general purpose global optimization software package," *Journal of global optimization* **8**, 201–205.

Sahinidis, N. V., 2019, "Mixed-integer nonlinear programming 2018," *Optimization and Engineering* **2**, 301–306.

SAS/OR. "SAS/OR 15.1 User's Guide Mathematical Programming," , 2019. URL https://support.sas.com/documentation/onlinedoc/or/151/ormpug.pdf

Sharma, M., Hahn, M., Leyffer, S., Ruthotto, L., and van Bloemen Waanders, B., 2020a, "Inversion of convection-diffusion equation with discrete sources," *Optimization and Engineering*

Sharma, M., Palkar, P., and Mahajan, A., 2020b, *Linearization and Parallelization Schemes for Convex Mixed-Integer Nonlinear Optimization*, Tech. Rep. 7793

Shinano, Y., 2018, "The ubiquity generator framework: 7 years of progress in parallelizing branch-and-bound," in *Operations Research Proceedings 2017* (Springer). pp. 143–149.

Shinano, Y., Achterberg, T., Berthold, T., Heinz, S., and Koch, T., 2011, "ParaSCIP: a parallel extension of SCIP," in *Competence in High Performance Computing 2010* (Springer). pp. 135–148.

Shinano, Y., Achterberg, T., Berthold, T., Heinz, S., Koch, T., and Winkler, M., 2016, "Solving open MIP instances with ParaSCIP on supercomputers using up to 80,000 cores," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (IEEE). pp. 770–779.

Shinano, Y., Berthold, T., and Heinz, S., 2018, "ParaXpress: an experimental extension of the FICO Xpress-Optimizer to solve hard MIPs on supercomputers," *Optimization Methods and Software* **33**, 530–539.

Shinano, Y., and Fujie, T., 2007, "ParaLEX: A parallel extension for the CPLEX mixed integer optimizer," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting* (Springer). pp. 97–106.

Shinano, Y., Heinz, S., Vigerske, S., and Winkler, M., 2017, "FiberSCIP - a shared memory parallelization of SCIP," *INFORMS Journal on Computing* **30**, 11–30.

Shinano, Y., Rehfeldt, D., and Gally, T., 2019, "An easy way to build parallel state-of-the-art combinatorial optimization problem solvers: A computational study on solving steiner tree problems and mixed integer semidefinite programs by using ug [SCIP-*,*]-libraries," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (IEEE). pp. 530–541.

SHOT. "The Supporting Hyperplane Optimization Toolkit," , 2019. URL http://www.github.com/coin-or/shot

Smith, E. M., and Pantelides, C. C., 1997, "Global optimisation of nonconvex MINLPs," *Computers & Chemical Engineering* **21**, S791–S796.

Smith, L., Chinneck, J., and Aitken, V., 2013a, "Constraint consensus concentration for identifying disjoint feasible regions in nonlinear programmes," *Optimization Methods and Software* **28**, 339–363.

Smith, L., Chinneck, J., and Aitken, V., 2013b, "Improved constraint consensus methods for seeking feasibility in nonlinear programs," *Computational Optimization and Applications* **54**, 555–578.

Tawarmalani, M., and Sahinidis, N. V., 2005, "A polyhedral branch-and-cut approach to global optimization," *Mathematical Programming* **103**, 225–249.

Tawarmalani, M., and Sahinidis, N. V., 2013, *Convexification and global optimization in continuous and mixed-integer nonlinear programming: theory, algorithms, software, and applications*, Vol. 65 (Springer Science & Business Media).

Torczon, V., 1997, "On the convergence of pattern search algorithms," *SIAM Journal on optimization* **7**, 1–25.

Ugray, Z., Lasdon, L., Plummer, J., Glover, F., Kelly, J., and Martí, R., 2007, "Scatter search and local NLP solvers: A multistart framework for global optimization," *INFORMS Journal on Computing* **19**, 328–340.

Van Hentenryck, P., 1999, *The OPL optimization programming language* (MIT press).

Vigerske, S., and Gleixner, A., 2018, "SCIP: Global optimization of mixed-integer nonlinear programs in a branch-and-cut framework," *Optimization Methods and Software* **33**, 563–593.

Wächter, A., and Biegler, L. T., 2006, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical programming* **106**, 25–57.

Witzig, J., Berthold, T., and Heinz, S., 2017, "Experiments with conflict analysis in mixed integer programming," in *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems* (Springer). pp. 211–220.

Wolsey, L. A., and Nemhauser, G. L., 1999, *Integer and combinatorial optimization*, Vol. 55 (John Wiley & Sons).

Xpress. "FICO Xpress-Optimizer," , 2019. URL http://www.fico.com/en/Products/DMTools/xpress-overview/Pages/Xpress-Optimizer.aspx

Xu, Y., Ralphs, T. K., Ladányi, L., and Saltzman, M. J., 2009, "Computational experience with a software framework for parallel integer programming," *INFORMS Journal on Computing* **21**, 383–397.

Zhou, K., Chen, X., Shao, Z., Wan, W., and Biegler, L. T., 2014, "Heterogeneous parallel method for mixed integer nonlinear programming," *Computers & Chemical Engineering* **66**, 290–300.

# List of Publications, Posters and Talks

- Larson J., Leyffer S., Palkar, P. and Wild, S., 2017, "A Method for Convex Black-Box Integer Global Optimization", 2020, *(accepted in) Journal of Global Optimization*.
- Sharma M., Palkar P., and Mahajan A., "Linearization and Parallelization Schemes for Convex Mixed-Integer Nonlinear Optimization," *Under review, Optimization Online*, http://www.optimization-online.org/DB_FILE/2020/05/7793.pdf
- Palkar P., Mahajan A., "Addressing Anomalies In Parallel Branch- and-Bound Algorithms for Mixed-Integer Nonlinear Optimization", 2021, *(To be communicated)*.
- Palkar P., Mahajan A., "A Branch-and-Estimate Heuristic Procedure for Solving Nonconvex Integer Optimization Problems," *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 1143–1151.
- Liu Z., Rajkumar K., Leyffer S., Palkar, P. and Foster, I., 2017, "A Mathematical Programming-and Simulation-Based Framework to Evaluate Cyberinfrastructure Design Choices," *2017 IEEE 13th International Conference on e-Science (e-Science)*, 148-157.
- Accelerating LP, NLP and MILP Based Algorithms for Convex MINLPs using Parallelization Schemes. *52nd Annual Convention of ORSI & International Conference, IIM Ahmedabad, India*, December 15-18, 2019.
- Parallel Algorithms for Convex Mixed-Integer Nonlinear Programming. *3rd International Conference and Summer School on Numerical Computations: Theory and Algorithms (NUMTA) 2019, Calabria, Italy*, June 15-21, 2019.

- Optimizing Convex Derivative Free Functions Over Integer Lattice. *LANS Summer Argonne Student Symposium (SASSy) Part II, Argonne National Laboratory, Lemont, USA*, August 08, 2018.
- A Globally Convergent Simulation-based Optimization Algorithm with Integer Constraints. *23rd International Symposium on Mathematical Programming (ISMP), Bordeaux, France*, July 01-06, 2018.
- Mixed-Integer Derivative-Free Optimization. *Mixed Integer Programming (MIP) Workshop 2018, Greenville, SC, USA*, June 18-21, 2018.
- MINOTAUR: Mixed-Integer Nonlinear Optimization Toolkit - Algorithms, Underestimators, Relaxations. *OPTSUM 2017, Mumbai, India*, September 14, 2017.
- Parallel Branch-and-Bound Algorithms For Convex Mixed-Integer Nonlinear Programs. *ORSI-2016, New Delhi, India*, December 12-14, 2016.
- Towards global optimization of mixed integer nonlinear programming problems. *Innovation Day, JDA, Hyderabad, India*, April 06, 2016.

# Acknowledgements

First, I would like to thank my advisor Prof. Ashutosh Mahajan for always being supportive on both academic and personal fronts. The time spent with him during research as well as during various teaching assistantships and system admin activities has helped me a lot to develop and hone my technical and general problem solving skills.

I would also like to thank the faculty at IEOR for their direct and indirect inputs during this endeavour, by means of course instructions, interactions etc. Similarly, I would like to thank my fellow research scholars and colleagues at the IEOR office for always being supportive and helpful.

The work on Mixed-Integer Derivative-Free Optimization has been done in collaboration with Dr. Jeffrey Larson, Dr. Sven Leyffer and Dr. Stefan Wild, all from Argonne National Laboratory (ANL), Lemont, USA. [1] I would like to thank all of them for their valuable guidance on this part of the thesis, especially Sven, who provided me the opportunity to visit ANL in the summers of 2017 and 2018, which were quite enriching, and also for always being there to help whenever needed.

I am thankful to my family and friends for being supportive during my PhD and always. At last, I would like to thank my wife for always being there with me on this journey. Her presence made the sailing smooth and her contributions towards this thesis are inseparable from mine.

***Prashant Palkar***
IIT Bombay
September 26, 2022

[1] Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439; email ids: jmlarson@anl.gov, leyffer@anl.gov, wild@anl.gov, in respective order.