Les Cahiers du GERAD ISSN: 0711–2440

CCGO: Fast heuristic global optimization

M. Shafique,
J. W. Chinneck

G–2017–71

August 2017

GERAD HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

CCGO: Fast heuristic global optimization

Mubashsharul Shafique [a]

John W. Chinneck [a, b]

[a] *Carleton University, Ottawa (Ontario) Canada, K1S 5B6*

[b] *GERAD, HEC Montréal, Montréal (Québec), Canada, H3T 2A7*

chinneck@sce.carleton.ca

August 2017

Les Cahiers du GERAD

G–2017–71

Abstract: Global optimization problems are very hard to solve, especially when the nonlinear constraints are highly nonconvex, which can result in a large number of disconnected feasible regions. Complete methods based on spatial branch and bound can find a global optimum point, but can also be slow. Multi-start methods can be used instead when shorter solution times are important. These methods explore the variable space to identify promising points from which to launch a local solver. We present the CCGO multi-start heuristic which uses a combination of various types of initial point scatter, constraint consensus to move points toward feasibility, clustering to identify disconnected feasible regions, and simple search to improve point clusters, before selecting local solver launch points. It targets highly nonconvex models. This heuristic performs very well in comparison to existing commercial complete and multi-start solvers.

Keywords: Global optimization, nonlinear optimization, multi-start methods

Acknowledgments: The authors gratefully acknowledge the support for this research provided by NSERC Canada via a Discovery Grant to Professor Chinneck.

1 Introduction

Global optimization (GO) involves finding a best feasible solution to a model that has at least one nonlinear function among the constraints and/or objective function. In the most general case, the model has an objective function, functional constraints, and variable bounds. All variables are continuous, and together with simple upper and lower bounds on their values define an n-dimensional hyperbox called the *variable space*. A generic GO model is described in Equation (1).

$$\begin{aligned} &Min\ f(\boldsymbol{x}) \\ &g_i(\boldsymbol{x})\{\leq,=\}0 && for \quad i = 1 \dots m \\ &l_j \leq x_j \leq u_j && for \quad j = 1 \dots n \end{aligned} \tag{1}$$

Local optimization of a model of the type in Equation 1 requires only finding a solution that is feasible and best in a local neighbourhood, such as a small ball surrounding the solution point. Global optimization requires finding a best solution among all possible feasible solutions in the variable space. Without loss of generality we will assume minimization throughout the paper.

Global optimization is much harder than local optimization since it must overcome difficulties such as multiple local optima, disconnected feasible regions, etc. The nonconvexity of the feasible region, and possibly of the objective function, increases the challenge. Global optimization is NP-hard (Vavasis 1995), but is frequently required in engineering, finance, biology, etc. (Floudas 1999), hence practical solution algorithms and their software implementations are highly valued, especially as the scale and complexity of the models grow.

There are three main categories of solution methods.

Complete solvers such as BARON (Ryoo and Sahinidis 1996), Couenne (Belotti et al. 2009), and SCIP (Achterberg 2009) use spatial branching within a branch and bound framework to implicitly explore the entire variable space. Each node in the search tree requires the estimation of a lower bound on the value of the objective function in a sub region of the variable space. Considerable effort is often required to construct suitable underestimators that are as accurate as possible in each sub region. Complete solvers are very accurate, but are relatively slow, and do not scale well in general.

General metaheuristic methods such as simulated annealing (Kirkpatrick 1984) and tabu search (Glover and Laguna 1998) can be applied, generally by first converting the problem to an unconstrained form through the use of a penalty function for violating constraints. However the quality of the solutions returned by such methods is not very high.

Multi-start heuristics are the best alternative to complete solvers when the user can accept a potential loss in solution quality for a large gain in solution speed. In a naïve multi-start method, a local solver is launched from a large number of random points in the variable space. This can be done concurrently. In more advanced solvers such as MSNLP (Ugray et al. 2009) and AIMMS Multi-start (Hunting 2017) the variable space is searched in a more intelligent manner with the goal of launching a local solver from only a few well-chosen points since the local solution is the most computationally expensive step.

The Constraint Consensus Global Optimizer (CCGO) developed here is a multi-start solver. The algorithm is intended to produce good quality solutions (relatively close to the optimum objective function value) in a relatively short amount of time for large and highly nonconvex models, especially those having multiple disconnected feasible regions. This will be highly useful in practice when decisions must be made quickly. A good early incumbent solution is especially valuable for this reason. The best possible outcome would be to find optima as good as those found by complete solvers in times smaller than those required by existing commercial multi-start solvers. Making use of concurrency will help with this goal.

1.1 Multi-start solvers

An efficient multi-start solver launches the local solver from only a small number of points. The best possible solver would launch from exactly one point somewhere in the basin of attraction for a global optimum point. It is rarely possible to identify such an excellent launch point, so algorithms try to gain efficiency by not launching from points unlikely to lead to good solutions, and try not to select more than one point in each basin of attraction. Algorithms vary in how they address these issues.

The commercial solver Knitro (Byrd et al. 2006) launches its local solver from a large number of initial points within the variable bounds, or within a smaller user-specified hyperbox. It will launch from $min\{200, 10n\}$ points, where n is the number of variables, or a user-specified number.

For unconstrained models, the Multi-Level Single-Linkage algorithm (Ugray et al. 2009) scatters random points uniformly through the variable space, sorts them according to their objective function value, and selects a subset of the best points, excluding points that are within a specified distance from another point having a better objective function value. It may also eliminate points that are close to the variable space boundary, or that are within a critical distance to a previously found local minimum. The critical distance shrinks as points are added by further sampling, and various rules are used to gauge when to terminate the algorithm.

Random Linkage multi-start (Ugray et al. 2007) generates new random points on the fly and does not maintain a list of points. The local solver is launched from the new point with a probability based on its distance from the closest previous sample point with a better objective function value (the probability increases with distance).

MSNLP (Ugray et al. 2009) uses a similar approach for constrained problems, using a penalty function to evaluate points. It models basins of attraction as spheres in n-space. A list of random potential launch points is maintained. If a point is closer than a critical distance to a nearby point with a better penalty function value, then the merit filter prevents it from being used as a launch point. The critical distance is decreased as the algorithm proceeds, and a previously excluded potential launch point may be used in a subsequent cycle, based on a probability derived from its distance to the nearest point with a better penalty function value. The distance filter also skips potential launch points if they are within a critical distance of a previously found local optimum point.

The related solver OQNLP (Ugray et al. 2007) includes the MSNLP algorithms as well as an additional OptQuest driver to generate trial points. The OptQuest algorithm operates on an initial population of solutions, generating new solutions as combinations of members of the population. Thereafter it operates as in MSNLP.

The Aimms solver (Hunting 2017) has a multi-start option that maintains clusters of solutions. It starts with a uniformly distributed set of points within the variable space, each evaluated via a penalty function. Points may be passed to the local solver, and the output solutions are used to update the clusters and the cluster radii. Potential launch points within the cluster radii are deleted. As for the previous algorithm, the intent is to avoid launching the local solver multiple times within the same basin of attraction for the local solver.

The algorithm developed in this paper extends previous work by Smith et al. (2013a) that is described in Section 1.2.

1.2 Constraint Consensus in global optimization

Chinneck (2004) developed the *Constraint Consensus (CC)* algorithm as a way to quickly improve a user-provided launch point prior to launching a local solver; improvements were later made by Ibrahim and Chinneck (2008). Smith et al. (2013b) later developed the SUM variant which is used in this work. CC very rapidly moves from a given start point (possibly very far from feasibility) to a point that is close to a feasible region and greatly improves the success rate for local solvers in finding a feasible solution. It works by finding a *feasibility vector* for each violated constraint that connects the current point to the closest point

satisfying the violated constraint (exact for linear constraints, an approximation for nonlinear constraints), and then combining these into a single update *consensus vector* in various ways. The current point is updated by applying the consensus vector, and the process is repeated until a stopping condition such as acceptable closeness to feasibility, maximum iterations etc. is reached. CC is a variant of the well-known projection methods (Motzkin et al. 1954, Gubin et al. 1967, and Censor 1988) applied as a heuristic for sets of nonconvex nonlinear constraints.

MacLeod (2006) extended this idea to use CC to explore the variable space prior to launching a local solver. He divides each axis into zones and uses randomly launched CC solutions to reason about the likelihood of finding a feasible point in each zone. The outcome is a set of probabilities of local solver success assigned to each zone on each axis. Experimental results were promising.

The key theme in the multi-start methods described in Section 1.1 is an emphasis on where *not* to place local solver launch points. There has been relatively less work emphasizing where launch points *should* be placed. Early papers by Rinnooy Kan and Timmer (1987a, 1987b) showed that clustering methods, including single-linkage clustering, are especially efficient and accurate for identifying different regions for sampling, though their methods are not well-suited to identifying basins of attraction. It is useful to distinguish basins of attraction so that efficiency can be increased by launching a local solver exactly once in each basin.

Smith et al (2013a) describe an algorithm that attempts to identify basins of attraction. This is efficient, and is especially useful for complex models having multiple disconnected feasible regions. A key component of the algorithm is the use of the Constraint Consensus method to rapidly explore the variable space. The method by Smith et al. is specifically designed for large models having many variables and is designated within the paper as *MS+C* (for multi-start plus concentration). The key steps are:

1. *An initial sampling* of the variable space using Latin Hypercube Sampling (McKay et al. 1979). This ensures a good coverage of the variable space by a relatively small user-controlled number of sample points.
2. *Constraint consensus* concentration of the points. Running CC on each initial point moves it close to a feasible region (or more accurately, towards a point where the constraint violations are minimized).
3. *Choosing the critical distance.* The subsequent clustering step depends on the identification of a critical distance for isolating the individual clusters. The critical distance is found by calculating the inter-point distance for every pair of points returned in Step 2. A histogram of the inter-point distances is constructed automatically based on the smallest and largest inter-point distances found. Reasoning about peaks and valleys in the histogram yields a useful critical distance that usually corresponds to the distance between feasible regions.
4. *Clustering.* Single-linkage clustering (Gower and Ross 1969) begins by assuming that each point is its own cluster. The two clusters that are the closest are then merged if they are separated by less than the critical distance. The process continues until no more clusters can be merged.
5. *Choosing solver launch points.* Cluster points are CC end points, and may or may not be feasible. A single most promising point in each cluster is chosen and the local solver is launched from it. The most promising point is identified by the following hierarchy: feasible points first, with ties broken by the point having the better value of the objective function, followed by infeasible points having the smallest maximum constraint violation. The total number of local solver launches is determined by the number of clusters identified, which is not known in advance. However the promising points can themselves be arranged in a similar hierarchy and launched in that order until a pre-specified maximum number of local solver launches is reached.

Smith et al. (2013a) report a number of experiments that show that MS+C is very effective in identifying disconnected feasible regions when there are several of them. However when feasible regions are very large it may launch the local solver from multiple points in or near the same large feasible region. As for other methods, it may also launch from points near places that are close to local but positive minima in the sum of the constraint violations. Experimental results show that MS+C is quite successful in comparison to the commercial solver Knitro 6.0 (Byrd et al. 2006). It is much faster than Knitro, and finds more feasible

solutions, but finds fewer best solutions. A major goal of the extended algorithm described in this paper is to improve performance in seeking optimum solutions.

2 The CCGO heuristic for global optimization

The CCGO algorithm developed here extends the work by Smith et al. (2013a). A main theme of the extensions is improved sampling of the variable space, including the use of launch boxes, unequal Latin Hypercube sampling, weighted sampling, heuristics for finding sufficient evaluable points, multiple rounds of sampling, and techniques for selecting launch points from among the available candidate launch points to insure both good quality points and good coverage of the potential different basins of attraction. A second theme is better movement towards optima, which introduces the use of a penalty function and the simple search heuristic. Concurrency is used as much as possible to speed the solution. At a high level, CCGO has the steps shown above for the MS+C algorithm, with a number of important improvements in the details, and some differences in the overall flow, described later.

2.1 Initial sampling of the variable space

As in MS+C, the first step is to randomly sample within the variable space, selecting points from which to initiate the Constraint Consensus method. CCGO uses Latin Hypercube Sampling for better coverage of the complete range of points, but makes some improvements over MS+C.

First, the variable space may be very large, especially when no bounds (or very large bounds) are specified for some variables. Previous work (Lasdon et al 2004, MacLeod 2006, Lasdon and Plummer 2008) has shown that limiting the initial sampling to a much smaller space, typically $\pm 1 \times 10^2$ or $\pm 1 \times 10^4$ can improve local solver success, because the variables frequently take values in this range in model solutions (MacLeod 2006). CCGO uses this concept: its initial random sampling takes place within a launch box of size $\pm 2 \times 10^4$ (or suitably smaller to respect the variable bounds). The CC solution is not restricted to the launch box, and may move outside of it.

In *Latin Hypercube sampling (LHS)*, each axis is typically subdivided into zones of equal width. If there are p sample points, then each axis is divided into p zones of equal width. Sampling is random, but assures that each zone on each axis is sampled exactly once (McKay et al. 1979). A particular advantage of LHS is that the number of sample points does not depend on the number of variables in the model: it can be fixed at any desired value.

CCGO uses both *uniform* and *nonuniform* subdivision of the axes for LHS. During nonuniform LHS, CCGO subdivides the axes where possible into the ranges 0-0.1, 0.1-1, 1-10, 10-100, and uniformly thereafter depending on the number of sample points. Adjustments are made if the variable spans zero (see Shafique (2017) for details). Previous work (Ibrahim and Chinneck, 2008) has shown the value of starting points near zero, so nonuniform sampling makes sure that area is well-sampled. This is an improvement over uniform sampling, which may place no points near zero, especially if the variable bounds are large.

It is not unusual for the evaluation of a nonlinear function to fail at a random point (e.g. due to overflow or underflow), so this must be handled during sampling. Points that evaluate in all functions without error are termed *clean points* (those that fail to evaluate in at least one constraint are *unclean points*). CCGO must generate a specified minimum number of clean points before exiting the initial sampling phase. Every potential point must be evaluated in every constraint; if it fails when evaluated in a particular constraint, then the rest of the constraint evaluations can be skipped. It is thus efficient if the evaluation failure happens early in the list of constraints. CCGO encourages early failure by evaluating the point in the nonlinear constraints before the linear constraints, since the nonlinear constraints are more likely to cause evaluation failure. Further, CCGO notes which constraints have caused a previous evaluation failure and tries these first in subsequent point evaluations. It also makes note of the variables occurring in each constraint that fails to evaluate correctly.

Insufficient clean points may be generated in a single scatter, so scattering is repeated as needed; each subsequent scatter tries to generate the required number of remaining points. Two strategies speed the

exit from this loop. First, if there are no clean points after a scatter, then the launch box is adjusted by considering the variables in the failed constraints. The launch box for these variables is contracted towards zero in a manner that removes a quarter of their initial launch box range. The smaller new box is resampled. The contraction process continues until the needed points are generated, or a variable range is less than 10% of its original launch box range, at which point the sampling restarts with the original launch box dimensions.

Second, if there are clean points in the current scatter, but fewer than the required number, then the launch box is adjusted in a different way, taking advantage of the information gleaned from the locations of the clean points in this scatter. If there are at least two clean points, then the smallest hyperbox containing these points is constructed, and uniform Latin Hypercube sampling is applied in this hyperbox to generate more points (the smaller of the number of clean points in this scatter, or the required number of remaining points).

2.2 Weighted resampling of the variable space

In the overall algorithm, described in Section 2.4, there are multiple rounds, each beginning with a random sampling of the variable space. The subsequent resampling can take advantage of information generated during earlier sampling phases to weight the sampling away from areas that were previously sampled, or that are near previously discovered clusters, or that tended to cause constraint evaluation errors. For this reason, earlier sampling rounds remember information about the location of certain types of points, so that subsequent sampling has a lower probability of sampling near them. Point information is retained for all CC launch points (clean as well as unclean), failed points, and simple search (see Section 2.3) end points. The latter are included because it is likely that sampling near them will lead to more solutions in the same general area.

This list of points to avoid is saved and subsequently used to construct a new launch box having weighted sampling probabilities. The new launch box is equivalent to the original launch box or is suitably larger if any of the points in the list is outside the original launch box (simple search end points may be outside the launch box for example).

Uniform Latin Hypercube sampling is used during weighted resampling, but each zone in each dimension is assigned a different probability of sampling, depending on the number of points on the avoid list that fall into the zone. We denote the probability of sampling in a given zone k in a given dimension j as p_j^k and the number of points on the avoid list that fall in zone k of dimension j as f_j^k. Where there are n_{avoid} points in total on the avoid list, and n_{zones} zones, the sampling probability of zone k in dimension j is:

$$p_j^k = \frac{n_{avoid} - f_j^k}{n_{avoid}(n_{zones} - 1)} \tag{2}$$

Zones having fewer points from the avoid list will obviously have higher sampling probabilities.

2.3 Combined Objective Value and Simple Search

A deficiency of the Smith et al. (2013a) algorithm is that the most promising point in a cluster may not be near a good feasible local optimum point, and hence the local solver may not find the best local optimum in a given feasible region. This is especially true because it is unlikely that feasibility will be reached by a few steps of Constraint Consensus, and hence the cluster best point is likely to be simply the point that has the smallest maximum constraint violation, meaning that the objective value is not considered at all. Assuming the cluster of points is associated with a feasible region, it is thus desirable to search further within the feasible region to find a local solver launch point that is closer to the best local optimum in that feasible region.

It is also valuable to be able to compare points from different clusters. Thus if there is a limit on the number of local solver launches that can be made, the better points can be chosen for launch.

Resolving these issues requires an appropriate measure of the goodness of a point that incorporates both the objective function value and a measure of infeasibility. The CCGO algorithm uses a standard

penalty measure for this purpose, denoted as the *combined objective value (COV)*, and defined as follows for minimization:

$$COV(\boldsymbol{x}) = f(\boldsymbol{x}) + p(\boldsymbol{x})$$

Where $f(\boldsymbol{x})$ is the objective function value, and $p(\boldsymbol{x})$ is the penalty measure, defined as the square of the maximum constraint violation at the point $\boldsymbol{x}$.

Using COV as the measure of the value of a point, CCGO invokes *Simple Search (SS)* for each cluster of points. This is a population-based improvement heuristic that generates new points both within the space defined by the cluster of points, and exterior to it. If a newly generated point has a better COV value than the point having the worst COV value in the cluster, then it replaces the worst point. Thus the cluster as a whole may shrink around areas of good COV value, or may migrate towards areas having better values. Algorithm 1 provides the pseudocode for Simple Search.

If there are insufficient points in the cluster input to SS, then additional points are generated in the vicinity of the existing points (Step 1). Step 2.1 generates and evaluates a new random point that is interior to the cluster, while Step 2.2 generates and evaluates a new random point that may be exterior to the cluster. Note that there must be a difference between $COV(\boldsymbol{x}_{int})$ and $COV(\boldsymbol{x}_2)$ in Step 2.2, otherwise the exterior hyperbox cannot be constructed. If the two points have the same value of COV, then $\boldsymbol{x}_2$ is again chosen randomly from X_{ss} until we have two points with different values of COV.

INPUTS: (i) a set of points X_{in}, (ii) minimum improvement δ, (ii) maximum number of successive failures f_{tol}, (iv) minimum points required n_{min}, (v) critical distance d.

OUPUT: X_{ss}, a possibly improved set of points.

1. $X_{ss} \leftarrow X_{in}$. If $|X_{ss}| < n_{min}$, then create a sampling hyperbox and sample it uniformly to add sufficient additional points to X_{ss}. The sampling hyperbox is defined as follows: find the smallest hyperbox containing the points in X_{in}, expand it by d in every direction, and then shrink it as necessary to respect the original variable bounds.
2. Do until there are f_{tol} successive failures to find an improving point in either step:
   2.1. *Interior search*: choose two different random points in X_{ss}. Choose a random point $\boldsymbol{x}_{int}$ in the tightest hyperbox containing these two points. Find $\boldsymbol{x}_{worst}$, the point in X_{ss} that has the worst value of *COV*. If $COV(\boldsymbol{x}_{int}) + \delta < COV(\boldsymbol{x}_{worst})$, then replace $\boldsymbol{x}_{worst}$ by $\boldsymbol{x}_{int}$ and set the failure count to 0, else increment the failure count and set $\boldsymbol{x}_{int}$ to a random point in X_{ss}.
   2.2. *Exterior search*: Choose a random point $\boldsymbol{x}_2$ in X_{ss} that is not the same as $\boldsymbol{x}_{int}$. Comparing $\boldsymbol{x}_{int}$ and $\boldsymbol{x}_2$, call the point having the better *COV* value $\boldsymbol{x}_{better}$ and the other point $\boldsymbol{x}_{base}$. Generate the point $\boldsymbol{x}_{proj}$ by projecting from $\boldsymbol{x}_{base}$ through $\boldsymbol{x}_{better}$ as follows: $\boldsymbol{x}_{proj} = \boldsymbol{x}_{better} + (\boldsymbol{x}_{better} - \boldsymbol{x}_{base})$. Choose a random point $\boldsymbol{x}_{ext}$ in the tightest hyperbox containing $\boldsymbol{x}_{better}$ and $\boldsymbol{x}_{proj}$. Find $\boldsymbol{x}_{worst}$, the point in X_{ss} that has the worst value of *COV*. If $COV(\boldsymbol{x}_{ext}) + \delta < COV(\boldsymbol{x}_{worst})$, then replace $\boldsymbol{x}_{worst}$ by $\boldsymbol{x}_{ext}$ and set the failure count to 0, else increment the failure count if not already incremented in Step 2.1.
3. Return X_{ss}.

Algorithm 1: Simple Search

2.4 Multiple sampling rounds and choosing local solver launch points

CCGO uses a number of rounds to generate potential local solver launch points, as shown in Algorithm 2, which outlines the sequential version of the method. The best point from each cluster found in round r (as determined by the COV value) is stored in X^r_{best}, where the number of points in each round list depends on the number of clusters identified in that round. These lists of points are then processed to identify the set of local solver launch points X_{launch}.

The best point found during the sampling rounds in Step 1 identifies the best round r^*. The points in $X^{r^*}_{best}$ then form the core of X_{launch}, where the list is suitably shortened if there are too many points in it, or added to if there are too few points (Step 3). Note that there could be fewer than l_{max} points in X_{launch} if there are insufficient points in all other X^r_{best} to make up the difference. Note also that if points must

be added to X_{launch}, then we add the points that are farthest from those already in X_{launch} in an effort to increase the diversity of the launch points.

INPUTS: (i) maximum number of sampling rounds r_{max}, (ii) maximum number of local solver launches l_{max}.

OUTPUT: a solution point $\boldsymbol{x}^*$ or an error message.

1. Do r =1 to r_{max} times:
   1.1. Scatter CC initial points using LHS as described in Secs. 2.1 and 2.2.
   1.2. Run CC on each point in the initial scatter.
   1.3. Cluster the CC output points as in MS+C.
   1.4. For each cluster of points:
      1.4.1. Run simple search.
      1.4.2. Note the point having the best COV value in the cluster, and add it to X^r_{best}.
2. Identify the point having the best COV value among all points in all X^r_{best}. This identifies the best round, r^*.
3. $X_{launch} \leftarrow X^{r^*}_{best}$. If $|X_{launch}| > l_{max}$ then remove points from X_{launch} having the worst values of COV until n_{max} points remain, else if $|X_{launch}| < l_{max}$ then add points to X_{launch} by choosing sufficient points from any round other than r^* that are farthest from the best point in X_{launch}.
4. Sort the points in X_{launch} by ascending value of COV. Set the incumbent objective function value z^* to infinity.
5. For each point in X_{launch}, in order:
   5.1. Launch the local solver from the point.
   5.2. If the local solver solution is feasible and has a lower objective function value than the incumbent solution, then update the incumbent point $\boldsymbol{x}^*$ and z^*.
6. Return the incumbent point $\boldsymbol{x}^*$ and z^*, or an error message if no solution found.

Algorithm 2: Sequential CCGO.

Algorithm 2 does not specify what sort of sampling is done in Step 1. A possible arrangement might have 3 rounds of sampling, performing nonuniform LHS in round 1, uniform LHS in round 2, and weighted LHS in round 3. The relevant points from rounds 1 and rounds 2 would be saved to calculate the weighted probabilities for round 3

2.5 Concurrency

Concurrency incurs overheads in managing the simultaneous execution flows, so the benefits must outweigh the costs. For example, if a task has a small serial runtime, then running multiple such tasks concurrently may require more time and resources for handling the multiple flows than simply running all of the tasks serially. The main issues are what to run concurrently, and what degree of concurrency to use (i.e. the number of simultaneous operations). If there are u simultaneous execution units, then there should not be requests for more than u simultaneous operation flows, otherwise the timesharing among the flows will incur overheads due to context switching, cache contention, etc. Execution units are carefully managed throughout so that the number of execution flows in operation never exceeds the maximum number of execution units.

Concurrent CCGO uses the same general execution flow as in Algorithm 2. Concurrency is added where it has proved valuable:

- The CC runs in Step 1.2 are concurrent, with one execution flow for each CC start point.
- The simple searches in Step 1.4 are concurrent, with one execution flow for each cluster of points.
- The local solver launches in Step 5 are concurrent, with one execution flow for each local solver launch point. The list of launch points is constructed a little differently, as described below.

The Latin Hypercube Sampling in Step 1.1 is very fast and so is carried out sequentially, and the clustering in Step 1.3 is inherently sequential. All other steps not listed above are sequential.

The list of local solver launch points is constructed in a slightly different manner than for the sequential version of the algorithm. Possible launch points are collected in three different lists: a primary list X_p, a secondary list X_s, and a tertiary list X_t. The primary list X_p collects the best point in each cluster in each round, with all other cluster points added to X_t. If there are too many points in X_p (i.e. $|X_p| > l_{max}$) then the excess points with the worst COV values are moved into X_s. The points in all three lists are ordered from best to worst COV value. Local solver launches take points in order from X_p, X_s and then X_t as needed to provide l_{max} launch points. The goal of this scheme is to provide good quality start points (low COV) that are spatially diverse (hence one from each cluster in X_p and X_s). It also provides a long list of potential launch points in case l_{max} is large and a time limit is instead used to halt execution.

There is one additional concurrent element that occurs in a step that is not shown in Algorithm 2. In each round, at the conclusion of Step 1.2, the best CC output point is identified. If no incumbent solution has yet been found (and no local solver is currently running), then the local solver is launched from this point if an execution unit is available. It has been observed that this often yields an early high quality first incumbent solution.

Theoretically the largest number of execution units that can be usefully employed at any point in the algorithm is limited by the maximum of the number of CC, SS, and local solver runs. The number of CC runs is determined by the number of sample points, which is an input parameter; the number of clusters found will also be no larger than this. The number of local solver runs is another input parameter, but it is generally very small since local solver solutions are the most compute-intensive part of the algorithm. The most compute-intensive parts of the variable space exploration are the CC and SS runs (both very small compared to a local solver launch), which is where concurrent operation has the most impact.

3 The CCGO software

3.1 Software

The final version of CCGO is written in C++ and compiled using GCC 4.7.2 under Ubuntu Linux 11.10. The local solver is IPOPT 3.11.1 (Waechter and Biegler 2006), chosen because (i) it has relatively good performance compared to other local solvers, and (ii) its open source license allows running multiple copies concurrently as required by the CCGO algorithm. IPOPT was compiled with the linear solver MA86 (2016) for its effectiveness on large-scale linear systems. Concurrent execution is via *pthreads* of the POSIX Standard (POSIX 2016).

Major user-controlled parameters include (i) total number of points in the initial scatter, (ii) the CC parameters (algorithm variant, feasibility tolerance, movement tolerance, maximum iterations, time limit), (iii) maximum number of clusters, (iv) minimum points per cluster, and (v) the IPOPT parameters.

Input models are in the AMPL (Fourer et al. 1993) modelling language and are precompiled by AMPL into the *.nl* format. CCGO reads and modifies the *.nl* files directly using the AMPL-supplied AMPL-solver library interface. The *.nl* files must be modified to provide the local solver launch point.

3.2 Tuning experiments

A series of tuning experiments were performed on a tuning set of all 201 CUTEr (Gould et al. 2003) models having 300 or fewer constraints, and including at least one nonlinear constraint, or a subset of these. The serial version of the algorithm was used. Some parameters were fixed in all experiments:

- Constraint Consensus: the SUM variant with feasibility tolerance 10^{-6}, movement tolerance 10^{-6}, maximum 100 iterations and 1 second per run.
- Clustering: maximum 25 clusters.
- IPOPT: honor_original_bounds = yes, maximum 6000 iterations, maximum 60 CPU seconds. All other options set to default values.

The following parameters were tuned (tuned values shown in boldface):

- The form of the simple search penalty (square of the maximum constraint violation vs. **sum of the squares of the individual violations**).
- The simple search stopping conditions (2, **3**, or 4 consecutive failures to make an improvement greater than 10^{-6}, greater than 1, **greater than 5**, greater than 10, greater than 40%).
- Whether to consider **all constraints in CC and SS**, or only the nonlinear subset.
- Number of points in the initial scatter and number of rounds (120 points in 1 round, **60 points in each of 2 rounds**, 40 points in each of 3 rounds, 30 points in each of 4 rounds, 24 points in each of 5 rounds).
- Minimum number of points in any cluster in SS (5, **10**, 15).
- Number of local solver launches (1, **2**, 3, or 4).

Values for these parameters were initially fixed, and then varied one at a time, with the best values retained at each step. The main metrics were the quality of solutions returned (both first incumbent and final solution) and the speed of solution. See Shafique (2017) for details.

4 Experimental setup

4.1 Test models

The testing set consists of 94 CUTEr (Gould et al. 2003) models that have more than 300 constraints, including at least one nonlinear function (constraint or objective). There are 48 models in the linearly constrained (LC) set that have a nonlinear objective function and only linear constraints. There are 46 models in the nonlinearly constrained (NLC) set, each having at least one nonlinear constraint. The testing set is completely distinct from the tuning set of small models. Statistics for the models in the testing set are shown in Table 1.

Table 1: Test model statistics

	All			Linearly Constrained			Nonlinearly Constrained		
	Avg	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**
Variables	5959.6	3.0	20200.0	6774.5	20.0	20200.0	5109.2	3.0	20000.0
Constraints	4228.2	313.0	14000.0	4541.3	356.0	12000.0	3901.5	313.0	14000.0
NL Constraints	1588.8	0.0	10000.0	0.0	0.0	0.0	3246.7	249.0	10000.0

A subset of 15 of the NLC models are highly nonconvex, as empirically determined by the MProbe software (Chinneck 2001); these are listed in Table 2. Each of these models has a large number of nonquadratic nonlinear constraints that have a nonconvex region effect (e.g. a concave constraint of type $\leq$), making it probable that there are multiple disconnected feasible regions. Difficult large-scale models of this type are the main target of CCGO.

4.2 Comparison solvers

CCGO is compared to 6 state-of-the-art solvers, run on the same hardware. All solvers were run with default settings, subject to the 30 minute time limit. Three of the comparison solvers use multi-start heuristics and three are complete solvers. The modelling systems varied as required by the solvers. Note that presolving (such as available via AMPL) was not used, since it is not available in all of the modelling systems. Details are given in Table 3.

Table 2: MProbe statistics for the highly nonconvex models

Model	Variables	Constraints			Nonconvex Region Effect	
		Linear	Quadratic	General NL Nonlinear	Total	In General Nonlinear Constraint
ARTIF	5000	50	0	4950	4950	4950
BDVALUE	5000	0	0	5000	5000	5000
BRITGAS	450	0	24	336	360	360
CBRATU2D	882	0	0	882	882	882
CHEMRCTA	5000	4	0	4996	4996	4996
CHEMRCTB	1000	2	0	998	998	998
CORKSCRW	8997	6000	0	1000	1000	1000
CRESC132	6	0	0	2654	2654	2654
DTOC4	14996	5000	0	4997	4997	4997
OET2	3	2	0	1000	1000	1000
ORTHREGD	10003	0	0	5000	5000	5000
POROUS1	4900	0	0	4900	4900	4900
SAWPATH	589	586	1	195	195	195
SEMICON1	1000	0	0	1000	575	575
TRAINH	20000	5001	0	5001	5001	5001

Table 3: Comparison solvers

Solver	Type	Modelling System
Couenne 4.7 (Belotti et al. 2009)	Complete	AMPL
BARON 14.4 (Ryoo and Sahinidis 1996)	Complete	GAMS
SCIP 3.1.0 (Achterberg 2009)	Complete	AMPL
AimmsCmd 4.3.2.3 (Hunting 2017)	Heuristic	AIMMS
Knitro 9.0.1 Parallel Multi-start (Byrd et al. 2006)	Heuristic	AMPL
MSNLP 24.4.1 r50296 (Ugray et al. 2009)	Heuristic	GAMS

4.3 Metrics

Complete solvers are run once for each model. Multi-start heuristic solvers are run 5 times for each model and the median values (run time, solution value) are reported for comparison purposes. If one or more of the 5 runs fails, then the median solution value for the successful runs is reported. Failure is reported for a multi-start solver only when all 5 runs fail.

The solvers are compared against each other based on solution quality, runtime, and robustness for both first incumbent and final solution. To measure solution quality, results are compared to the best objective function value returned by any of the solvers in a given comparison set. Minor numerical differences are expected in the solutions provided by different solvers, so we measure the differences from the best solution and collect these into ranges; these results are provided in tabular form.

Solution speed is reported in wall clock time (in nanosecond resolution). Results are summarized in figures that report the fraction of the models solved (vertical axis) vs. solution time (horizontal axis). These figures also show the robustness of each solver by the height of the last point on the solver curve: the more robust the solver, the larger fraction of models it is able to solve.

4.4 Parameter settings

The parameter settings for CCGO are as described at the end of Section 3.2 with some exceptions listed here. The total runtime limit is 30 minutes. This is perhaps smaller than might be allowed in practice, but necessary due to the large number of experiments conducted. The hardware had 4 physical cores, so CCGO was limited to a maximum of 4 simultaneous threads. CCGO used two approaches to sampling: *Sample A* uses uniform LHS in the first round and nonuniform LHS in the second round; *Sample B* uses nonuniform LHS in the first round and weighted sampling in the second round. Two cores were devoted to Sample A and two to Sample B; each method independently selects local solver launch points. IPOPT default settings were used with the exception of the iteration count (set to 10^6 to make it virtually infinite).

Default parameter settings were used for all of the comparison solvers, except for the imposed 30 minute time limit. Note that involuntary termination at the time limit is handled in different ways by the various solvers: some delay termination until the next convenient moment after the time limit has been reached.

For AMPL-based solvers, the presolve option is turned off to maintain parity with solvers based on other modelling languages which do not have a presolver.

4.5 Hardware

Experiments were run on a 64-bit Fedora 17 system on a 3.4 GHz Intel i7-2600 processor (4 cores, 8 logical cores) with 16 Gb memory.

5 Experimental results

5.1 Linearly-Constrained models

The main target for CCGO is highly nonconvex nonlinear models having multiple disconnected feasible regions. However a surprisingly large number of the models in the CUTEr set are composed entirely of linear constraints coupled with a nonlinear objective function, and it is of interest to see how well CCGO fares on this class of models when compared with other solvers.

Table 4 summarizes the solution quality results to both the first incumbent and final solutions for the complete solvers vs. CCGO. The "Diff" column shows the range of differences from the best objective function value returned by any of the 4 compared solvers; the body of the table gives the count of the number of models having a difference in that range. Best results in the first and last row are shown in boldface. CCGO performs surprisingly well in terms of the solution quality of both the first solution and the final solution, having the largest number of solutions within 0.1 of the best solution found by any of the 4 compared solvers. It also never fails to find a solution for any of the 48 models in this group.

Table 4: LC models, complete solvers: solution quality

Diff	First incumbent				Final soln (completed or at timeout)			
	CCGO	**BARON**	**SCIP**	**Couenne**	**CCGO**	**BARON**	**SCIP**	**Couenne**
≤ 0.1	**39**	19	2	31	**37**	34	6	31
≤ 1	1	2	0	1	1	1	0	1
≤ 10	1	0	3	0	0	2	3	0
≤ 100	2	2	1	1	3	4	1	1
> 100	5	25	32	5	7	7	28	5
Failed	**0**	**0**	10	10	**0**	**0**	10	10

CCGO is somewhat slower to reach the first incumbent as compared to the complete solvers on the LC set; this is not surprising since it doesn't use linear computations (until the linear solver that is part of IPOPT is invoked). However solution times are very consistent: a first incumbent is reached for every model well within 1 minute. See Figure 1.

Figure 2 shows the times needed to reach the final solutions. The spikes for the complete solvers are due to their return of a solution at (or in some cases just past) the 30 minute time limit. CCGO has a relatively constant solution time whereas solution times can vary markedly for the complete solvers. SCIP and Couenne fail for about 20% of the models, and find lower quality solutions.

AIMMS finds the best quality solutions among the multi-start heuristic solvers on the LC set, as shown in Table 5. CCGO solution quality is in a similar range to Knitro and MSNLP for the first incumbent solution. None of the heuristic solvers fails to return a solution for any model within the time limit.

For the multi-start heuristic solvers, the first incumbent solution is very frequently the same as the final solution returned, indicating that it is generally safe to terminate CCGO after the first incumbent solution

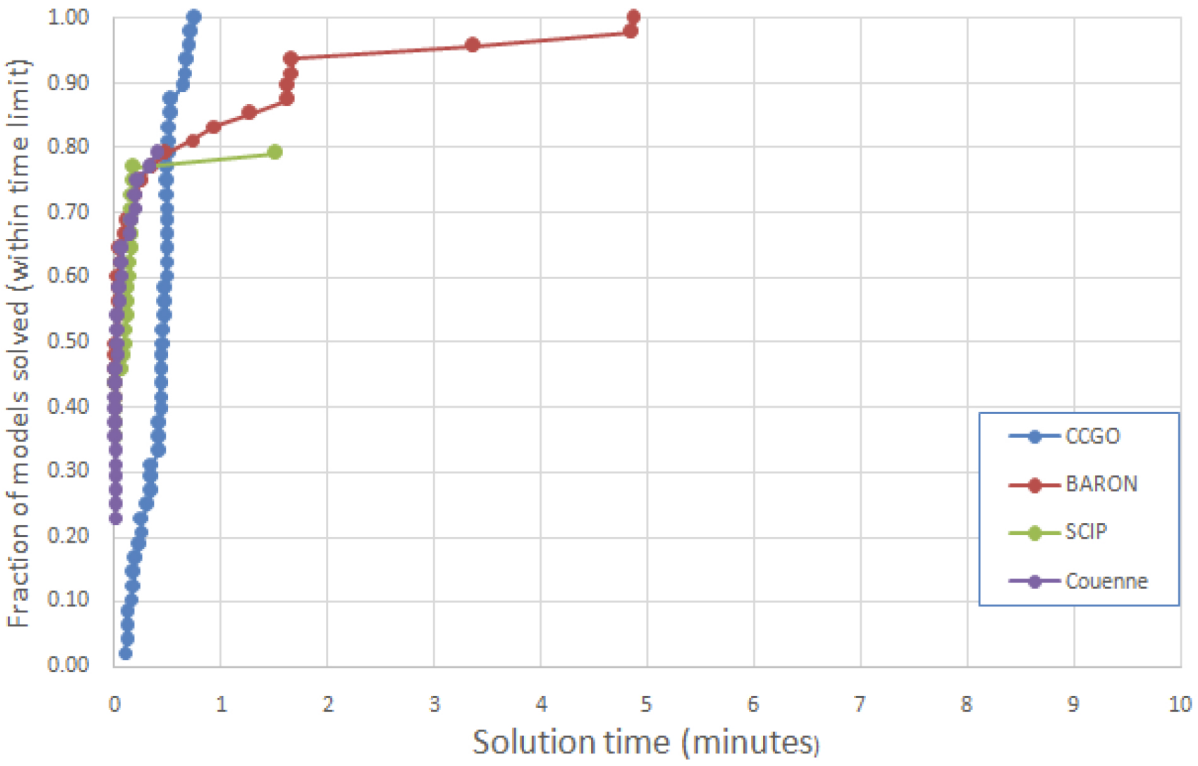

Figure 1: LC models, complete solvers: first incumbent solution times

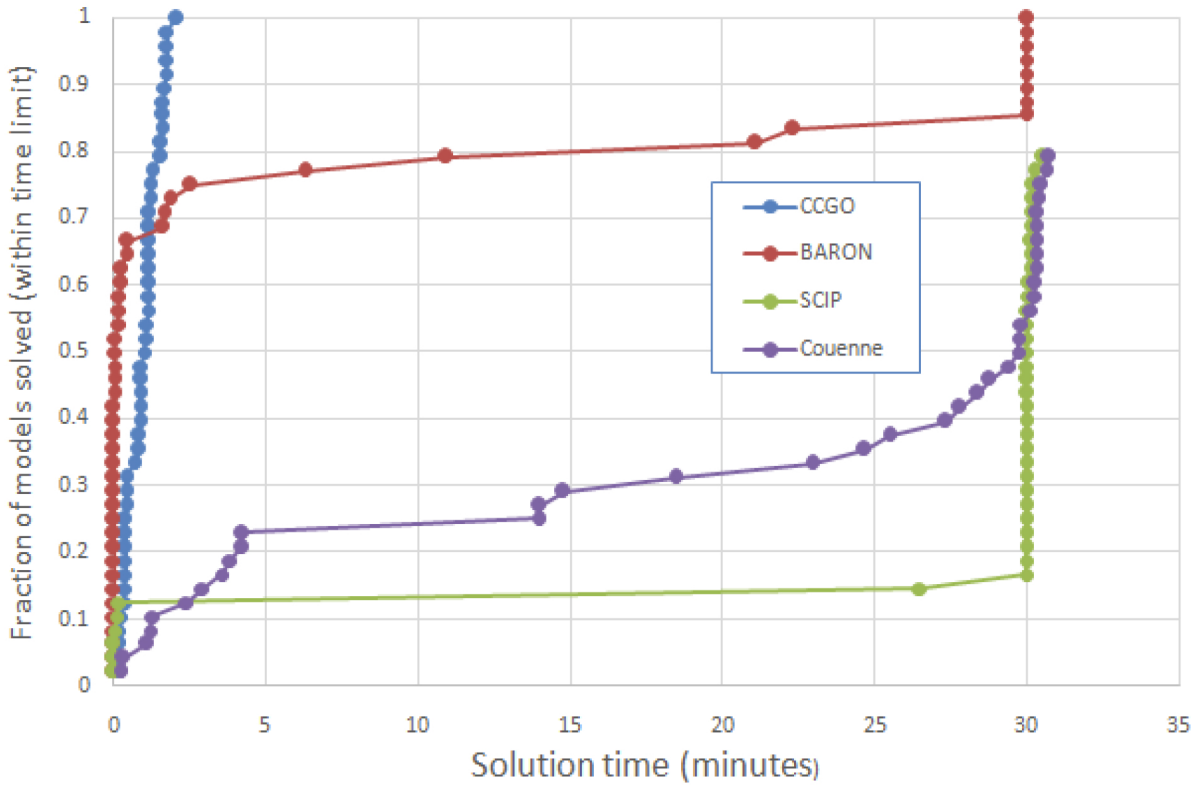

Figure 2: LC models, complete solvers: final solution times

Table 5: LC models, heuristic solvers: solution quality

Diff	First incumbent				Final soln (completed or at timeout)			
	CCGO	Knitro	MSNLP	AIMMS	CCGO	Knitro	MSNLP	AIMMS
≤ 0.1	25	26	28	**38**	23	31	28	**38**
≤ 1	2	2	1	0	2	1	1	0
≤ 10	1	2	1	0	0	1	1	0
≤ 100	3	3	1	2	4	3	2	2
> 100	17	15	17	7	19	12	16	7
> 100	**0**	**0**	**0**	1	**0**	**0**	**0**	1

is returned for a linearly constrained model. See Table 6. The complete solvers are more often continuing to improve their results until termination.

CCGO succeeds on all 5 runs for every model. In fact both Sample A and Sample B succeed on all 5 runs for every model. The variance in solution values and runtimes is very small over all 5 runs.

Table 6: LC models: first incumbent vs. final solution

	CCGO	BARON	SCIP	Couenne	Knitro	MSNLP	AIMMS
Solved of 48	**48**	**48**	38	38	**48**	**48**	47
Same solution	46	26	20	38	41	**48**	42
Same/solved (%)	95.8	54.2	52.6	**100.0**	85.4	**100.0**	89.4

Solution times for the heuristic solvers are plotted in Figure 3. Note the consistent run times for CCGO: all solutions complete within 1 minute. CCGO does surprisingly well on the LC models, considering that it is designed for highly nonconvex models having multiple disconnected feasible regions. The LC models have a single feasible region, so only the simple search routine in CCGO works towards improving the objective function before launching the local solver.

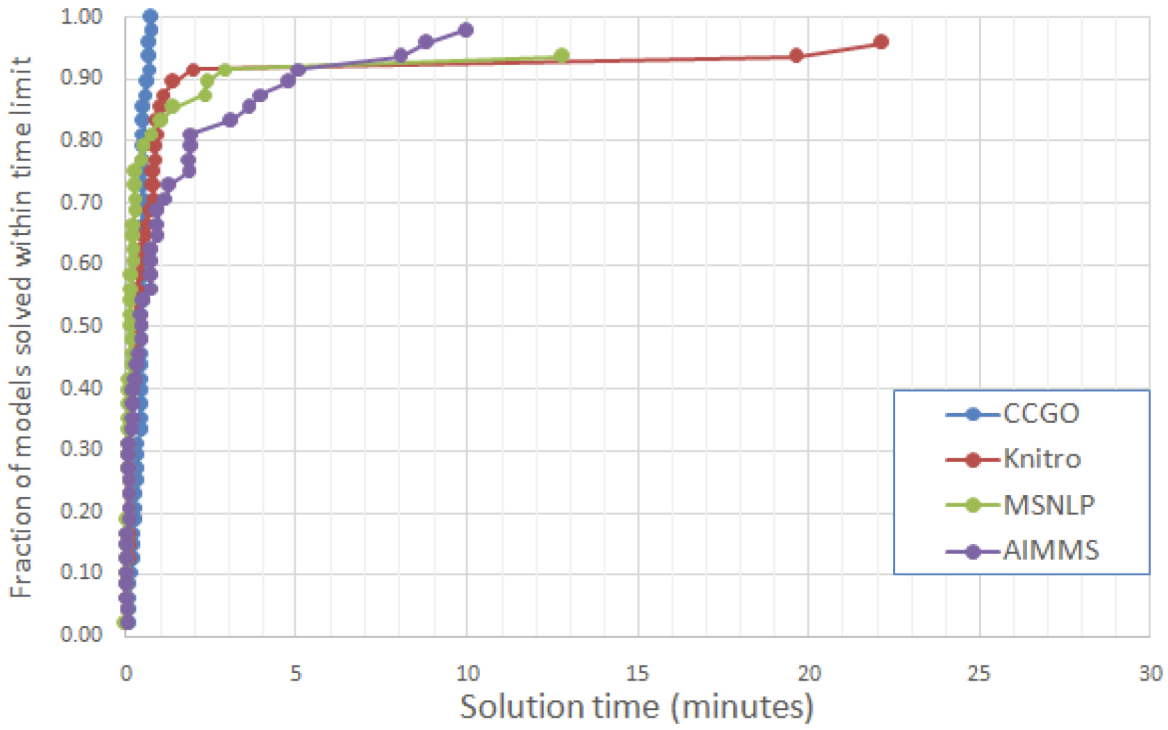

Figure 3: LC models, heuristic solvers: final solution times

5.2 Nonlinearly-Constrained models

Nonlinearly constrained models are the key focus for CCGO, especially difficult highly nonconvex models. This section analyzes the performance of CCGO on the 46 models having at least one nonlinear constraint; section 5.3 focuses on a highly nonconvex subset of 15 of these.

Table 7 summarizes the solution quality results for CCGO vs. the complete solvers on the nonlinearly constrained models. CCGO provides the best results, finding more solutions close to the best returned by any of the competing solvers, for both the first incumbent and final solutions. It is also the most robust solver, failing on only a single model.

Table 7: NLC models, complete solvers: solution quality

Diff	First incumbent				Final soln (completed or at timeout)			
	CCGO	**BARON**	**SCIP**	**Couenne**	**CCGO**	**BARON**	**SCIP**	**Couenne**
≤ 0.1	**40**	34	2	30	**37**	36	3	31
≤ 1	0	1	5	0	0	0	3	0
≤ 10	1	0	0	1	2	1	0	0
≤ 100	0	1	0	0	1	1	1	0
> 100	4	3	8	3	5	1	8	3
Failed	**1**	7	31	12	**1**	7	31	12

Table 8 summarizes the solution quality results for CCGO vs. the other multi-start heuristic solvers. CCGO provides solutions (first incumbent and final) of quality comparable to the three other heuristic solvers, all of which are commercial products of long standing. Further, CCGO is the most robust of the heuristic solvers, returning a solution for all but a single model.

Table 8: LC models, heuristic solvers: solution quality

Diff	First incumbent				Final soln (completed or at timeout)			
	CCGO	**Knitro**	**MSNLP**	**AIMMS**	**CCGO**	**Knitro**	**MSNLP**	**AIMMS**
≤ 0.1	**38**	**38**	**38**	**38**	38	**40**	37	37
≤ 1	0	1	0	0	1	0	0	0
≤ 10	1	0	0	0	0	0	0	0
≤ 100	1	0	0	0	1	0	0	0
> 100	5	4	1	1	5	3	2	2
Failed	**1**	3	7	7	**1**	3	7	7

Table 9 illustrates the value of having two separate methods of sampling the variable space. It is often the case that when one of the sampling methods fails to provide a useful launch point, the other one does.

Table 9: NLC models: combined robustness of CCGO sampling methods

	Number of models (of 46 total) having this many successes out of 5					
	0	**1**	**2**	**3**	**4**	**5**
Sample A	3	0	1	3	3	36
Sample B	7	0	2	3	2	32
Combined	1	0	1	1	4	39

As for the linearly constrained set, the first incumbent solution is almost always the same as the final solution for CCGO and the other multi-start heuristic solvers. This again suggests that it is safe to quit as soon as CCGO returns its first solution in most cases. See Table 10.

Table 10: NLC models: first incumbent vs. final solution

	CCGO	**BARON**	**SCIP**	**Couenne**	**Knitro**	**MSNLP**	**AIMMS**
Solved of 46	**45**	39	15	34	43	39	39
Same solution	**44**	35	9	31	41	39	39
Same/solved (%)	97.8	89.7	60.0	91.2	95.3	**100.0**	**100.0**

Figure 4 shows the final solution times for CCGO vs. the complete solvers. As before, the spike of results around (or slightly after) the 30 minute time limit is due to the complete solvers returning their incumbent result at this time. CCGO is able to solve 45 of the 46 models in less than 17 minutes, while the complete solvers frequently do not return a solution until near the time limit. CCGO is also the most robust solver.

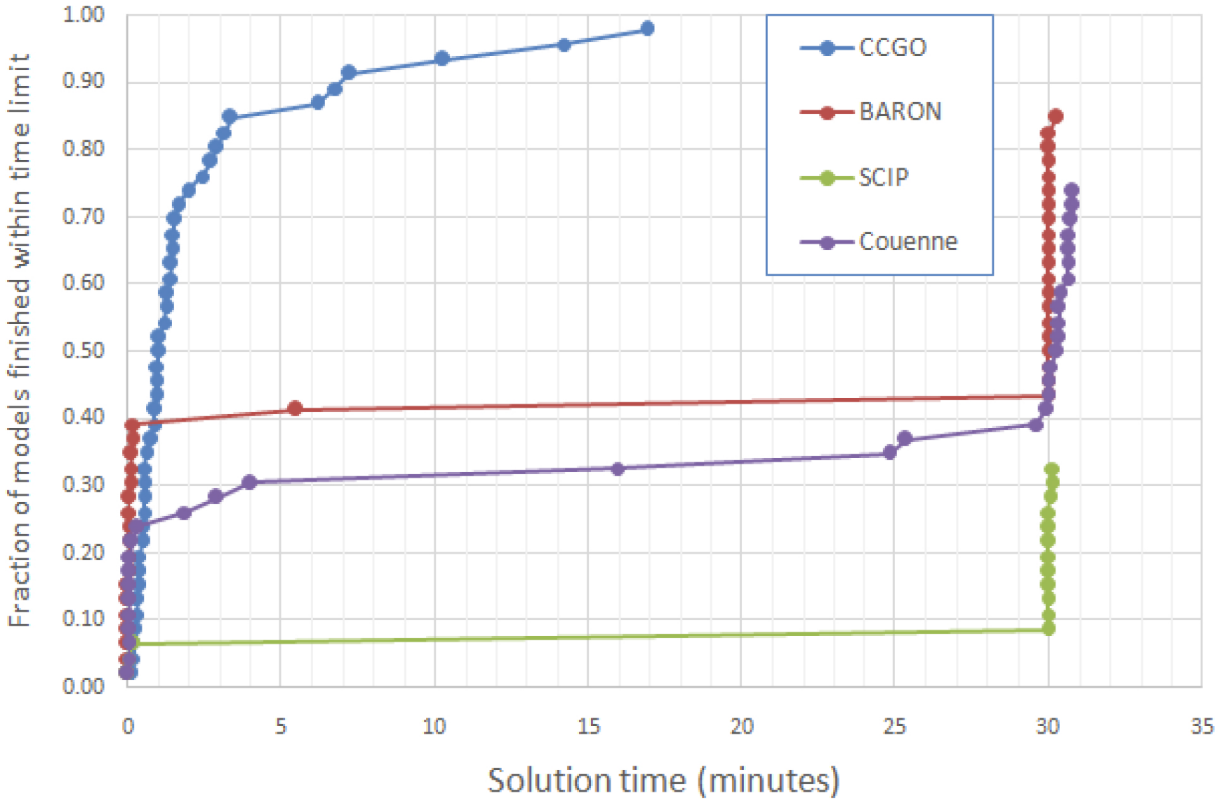

Figure 4: NLC models, complete solvers: final solution times

Figure 5 shows that CCGO returns a first incumbent solution quickly, with relatively constant solution times. CCGO finds a first incumbent solution for around 90% of the models within 1 minute of solution time, and for the remainder within about 9 minutes.

Figure 6 compares final solution times for the multi-start heuristic methods. As usual there are spikes as some of the methods return a solution at or even well past the time limit. CCGO is the fastest method by a significant margin. However all of the heuristic methods are generally similar in time to the first incumbent solution for most models, as shown in Figure 7.

5.3 Highly nonconvex feasible regions

Table 11 summarizes the solution quality results for the complete solvers over the subset of 15 highly nonconvex nonlinear models. CCGO provides the best results by a wide margin, most often finding solutions closest to the best returned by any solver in the group, with no failures. BARON provides the next best results, but fails on a third of the models.

Table 12 compares the quality of the solutions returned by the multi-start heuristic solvers. All of these methods perform very well as compared to the complete solvers; CCGO and Knitro return the best results and have no failures

Figure 8 shows the final solution times for CCGO vs. the complete solvers. CCGO is much faster than the complete solvers, and has much more consistent solution times, completing all solutions within a little

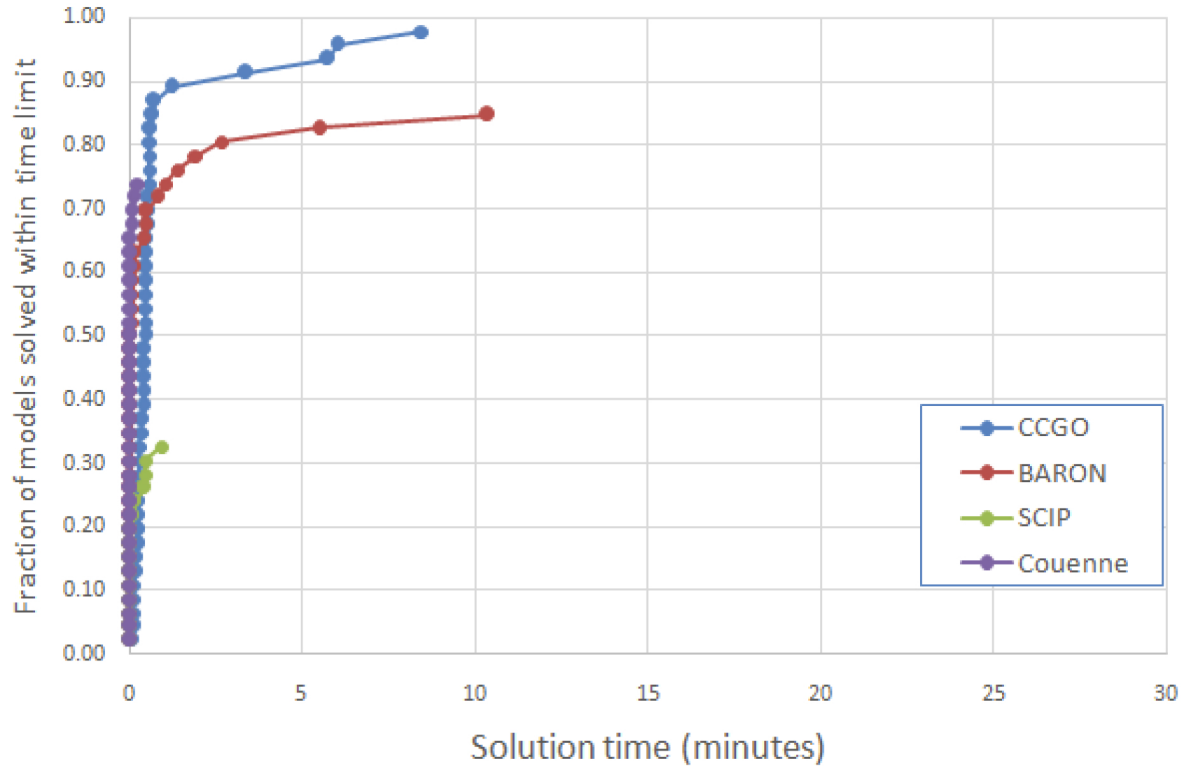

Figure 5: NLC models, complete solvers: first incumbent solution times

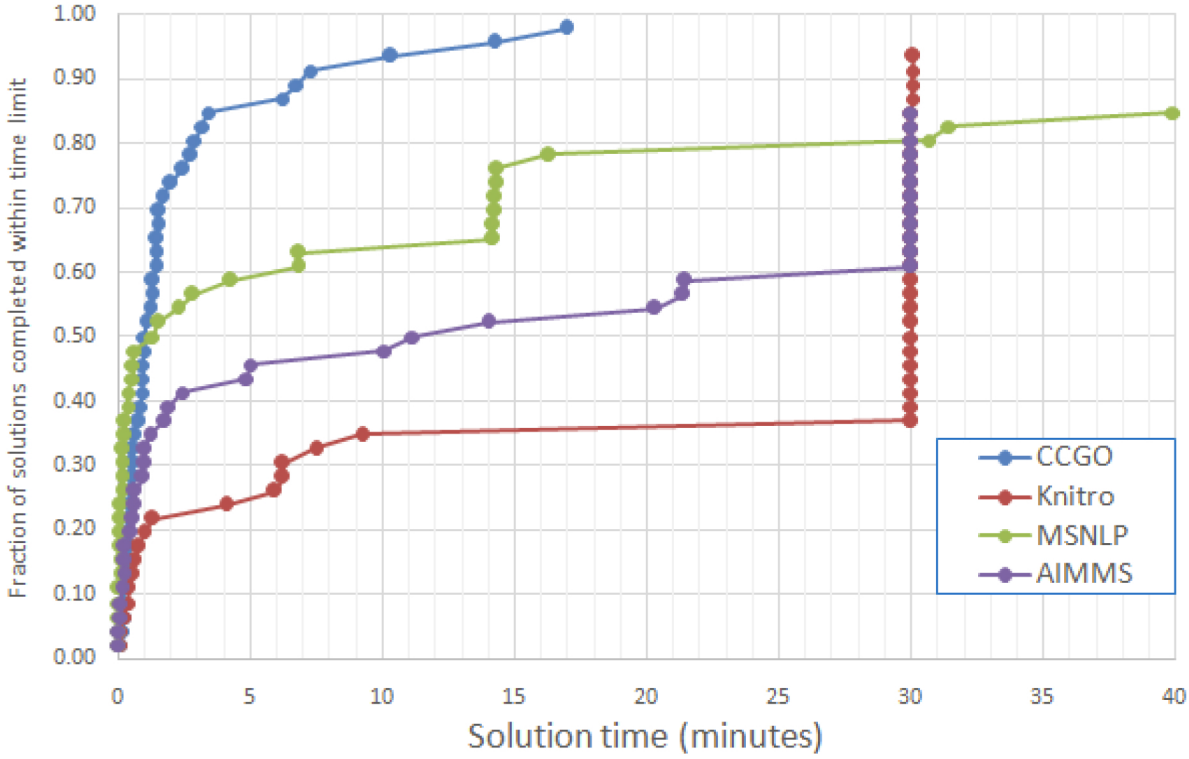

Figure 6: NLC models, heuristic solvers: final solution times

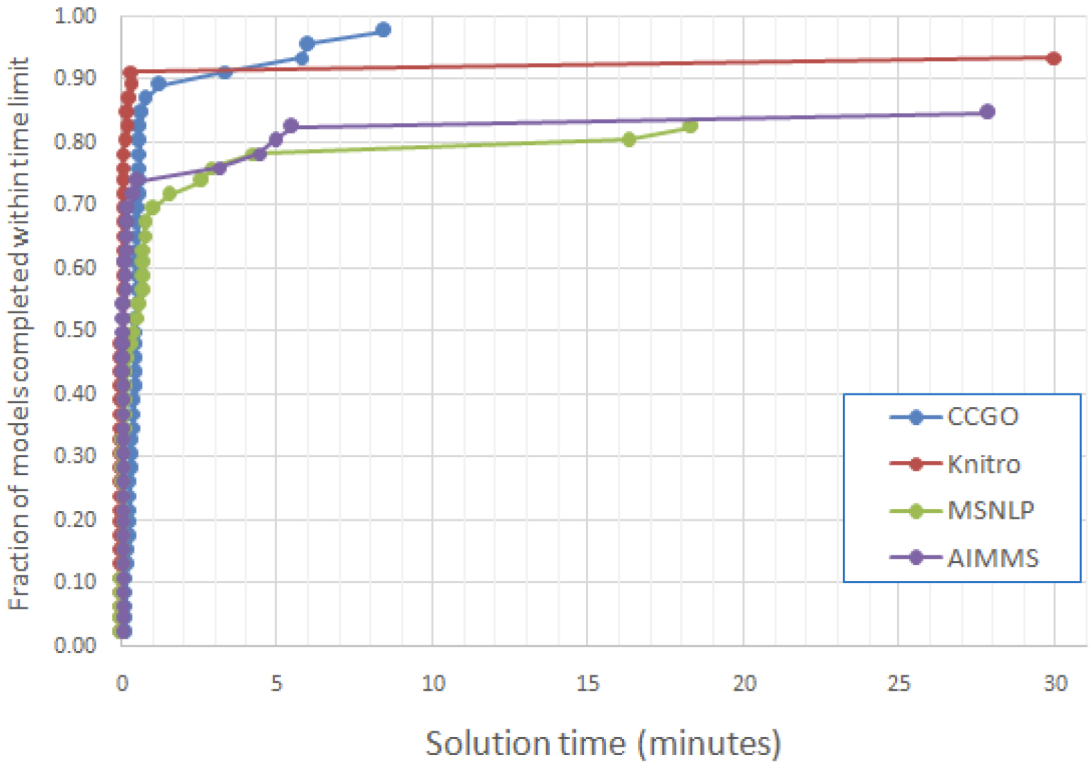

Figure 7: NLC models, heuristic solvers: first incumbent solution times

Table 11: Highly nonconvex models, complete solvers: solution quality

Diff	First incumbent				Final soln (completed or at timeout)			
	CCGO	BARON	SCIP	Couenne	CCGO	BARON	SCIP	Couenne
≤ 0.1	**14**	10	2	7	**14**	10	2	8
≤ 1	0	0	0	0	0	0	0	0
≤ 10	0	0	0	0	0	0	0	0
≤ 100	0	0	0	1	0	0	0	0
> 100	1	0	1	1	1	0	1	1
Failed	**0**	5	12	6	**0**	5	12	6

Table 12: Highly nonconvex models, heuristic solvers: solution quality

Diff	First incumbent				Final soln (completed or at timeout)			
	CCGO	Knitro	MSNLP	AIMMS	CCGO	Knitro	MSNLP	AIMMS
≤ 0.1	**14**	**14**	13	13	**14**	**14**	13	13
≤ 1	0	0	0	0	0	0	0	0
≤ 10	0	0	0	0	0	0	0	0
≤ 100	0	0	0	0	0	0	0	0
> 100	1	1	0	0	1	1	0	0
Failed	**0**	**0**	2	2	**0**	**0**	2	2

over 10 minutes. The story is similar for the first incumbent solution times, with CCGO returning most solutions within about 30 seconds, and all solutions within about 6 minutes.

Figure 9 shows the overall solution times for the multi-start heuristic solvers, with the usual spike in solutions returned around (or after) the 30 minute time limit. CCGO is fastest, completing all solutions within about 10 minutes. Knitro excels in returning first incumbent solutions quickly, returning solutions for all models within 30 seconds. CCGO provides the next best results, finding incumbent solutions within 30 seconds for all but two models.

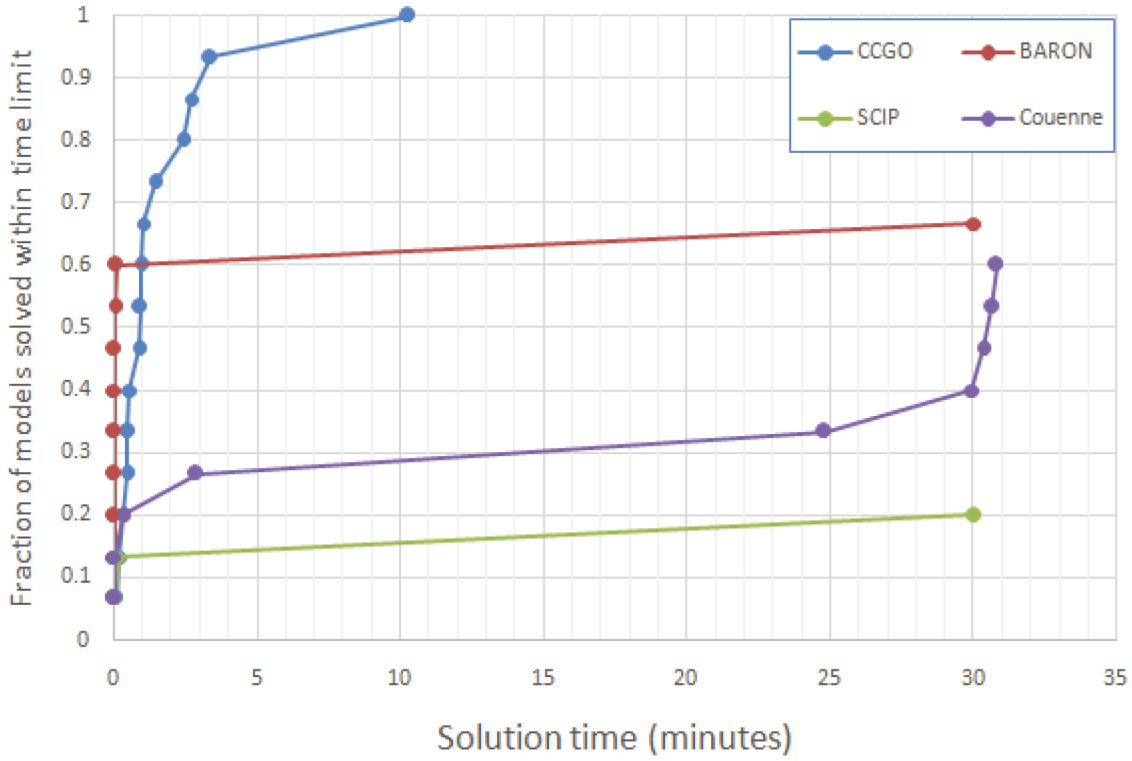

Figure 8: Highly nonconvex models, complete solvers: final solution times

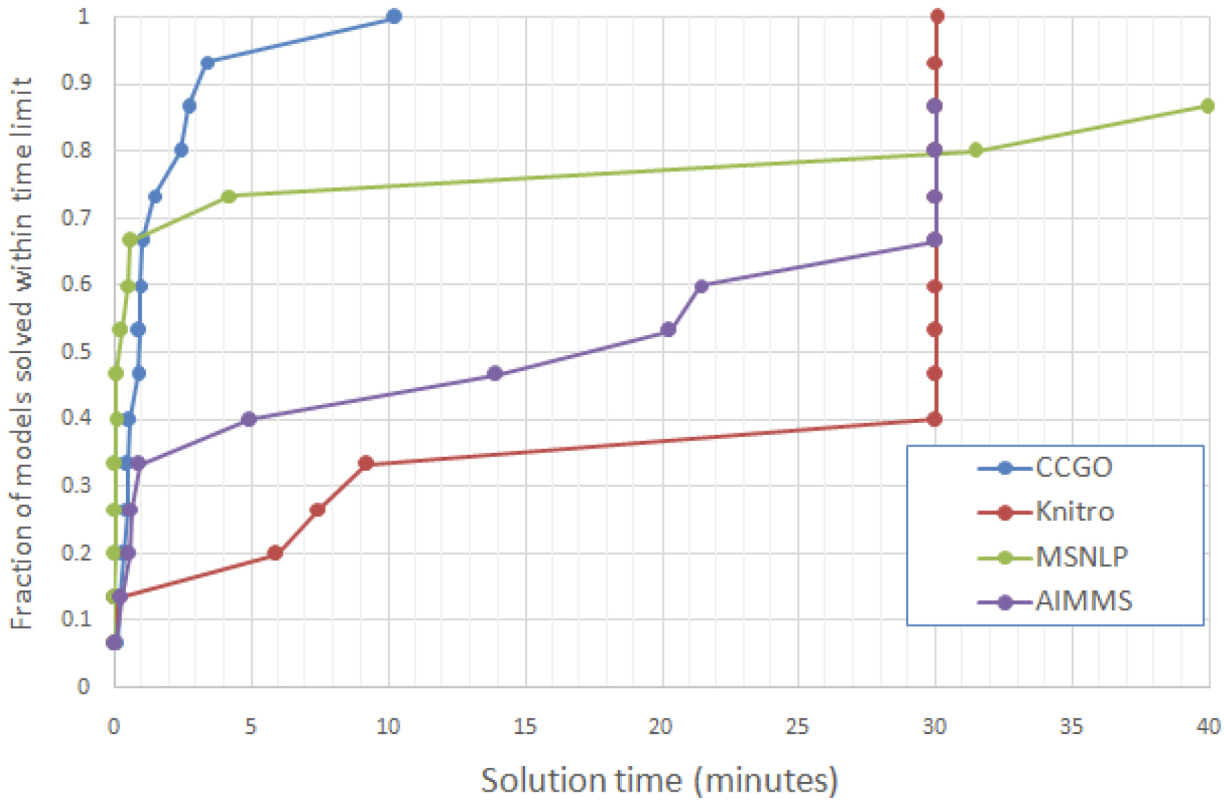

Figure 9: Highly nonconvex models, heuristic solvers: final solution times

As expected, CCGO is very effective for highly nonconvex nonlinear models, providing high quality solutions more quickly than both competing state-of-the-art commercial multi-start heuristic solvers and complete solvers.

6 Conclusions

The CCGO algorithm is an effective solution method for global optimization problems, especially the large-scale highly nonconvex models that are its particular focus. The CCGO solver implementation compares well to existing state-of-the-art commercial solvers. As for other multi-start heuristics, the focus is on quickly surveying the variable space to identify promising points from which to launch a local solver. CCGO identifies such points using a unique combination of multiple types of initial point scatter, constraint consensus to move points close to feasibility, clustering to identify distinct feasible regions, and simple search to improve both feasibility and objective function value. It combines this with methods for generating "clean points", multiple sampling rounds, secondary clustering, and concurrent implementation.

There are a number of simple modifications that could improve the performance of the current CCGO implementation even further:

- The current software does not use the AMPL nonlinear presolver. This would improve the results by tightening the variable bounds and perhaps eliminating constraints and variables. The resulting model would be simpler to solve.
- We use the open-source IPOPT solver for cost and licensing reasons. While it is a capable local solver, it may be that a different local solver could provide better results under some conditions.
- The first incumbent solution returned by CCGO is frequently identical to the final solution returned some time later. Thus it is possible to halt CCGO after the first solution is returned, thereby shortening the solution time.

There are many avenues for future improvements. The code would likely benefit from a full parameter tuning, and further exploration of the best ways to integrate the various methods of generating the initial point scatter.

References

T. Achterberg (2009). "SCIP: Solving Constraint Integer Programs", Mathematical Programming Computation, 1(1), 1–41.

P. Belotti, J. Lee, L. Liberti, F. Margot, and A. Waechter (2009). "Branching and Bounds Tightening Techniques for Non-Convex MINLP", Optimization Methods and Software, 24(4–5), 597–634.

R.H. Byrd, J. Nocedal, and R.A. Waltz (2006). "KNITRO: an Integrated Package for Nonlinear Optimization," in Large-scale Nonlinear Optimization, eds G. Di Pillo and M. Roma, 35–59, Springer.

Y. Censor (1988). "Parallel Application of Block-Iterative Methods in Medical Imaging and Radiation Therapy," Mathematical Programming, 42(1–3), 307–325.

J.W. Chinneck (2001). "Analyzing Mathematical Programs Using MProbe", Annals of Operations Research, 104(1), 33–48.

J.W. Chinneck (2004). "The Constraint Consensus Method for Finding Approximately Feasible Points in Nonlinear Programs", INFORMS Journal on Computing, 16(3), 255–265.

C.A. Floudas (1999). "Deterministic Global Optimization: Theory, Methods, and Applications", Springer Science+Business Media, Dordrecht.

R. Fourer, D.M. Gay, and B.W. Kernighan (1993). "AMPL: a Modeling Language for Mathematical Programming", Boyd and Fraser.

F. Glover, M. Laguna (1998). "Tabu Search", in Handbook of Combinatorial Optimization, vol. 3, eds. D.-Z. Du and P.M. Pardalos, pp. 2093–2229, Kluwer Academic Publishers.

N.I.M. Gould, D. Orban, and P.L. Toint (2003). "CUTEr and SifDec: a Constrained and Unconstrained Testing Environment, Revisited," ACM Transactions on Mathematical Software, 29(4), 373–394.

J.C. Gower and G. Ross (1969). "Minimum Spanning Trees and Single Linkage Cluster Analysis,", Applied Statistics, 18(1), 54–64.

L. Gubin, B. Polyak, and E. Raik (1967). "The Method of Projections for Finding the Common Point of Convex Sets", USSR Computational Mathematics and Mathematical Physics, 7(6), 1–24.

M. Hunting (2017). "A New Multistart Algorithm", https://aimms.com/files/7614/9020/3810/Multistart.pdf, accessed July 28, 2017.

W. Ibrahim and J.W. Chinneck (2008). "Improving Solver Success in Reaching Feasibility for Sets of Nonlinear Constraints", Computers and Operations Research, 35, 1394–1411.

S. Kirkpatrick (1984). "Optimization by Simulated Annealing: Quantitative Studies", Journal of Statistical Physics, 34(5–6), 975–986.

L. Lasdon and J.C. Plummer (2008). "Multistart Algorithms for Seeking Feasibility",Computers and Operations Research, 35(5) 1379–1393.

L. Lasdon, J.C. Plummer, Z. Ugray, and M. Bussieck (2004). "Improved Filters and Randomized Drivers for Multi-Start Global Optimization", technical report IROM-06-06, McCombs School of Business, the University of Texas at Austin.

MA86 (2016). "HSL MA86 v1.6.0 – C Interface", documentation date: October 3, 2016. http://www.hsl.rl.ac.uk/specs/hsl_ma86.pdf, accessed July 28, 2017.

M. MacLeod (2006). "Multistart Constraint Consensus for Seeking Feasibility in Nonlinear Programs", MASc Thesis, Systems and Computer Engineering, Carleton University, Ottawa, Canada.

M.D. McKay, R.J. Beckman, and W.J. Conover (1979). "Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output From a Computer Code," Technometrics, 21(2), 239–245.

T. Motzkin and I. Schoenberg (1954). "The Relaxation Method for Linear Inequalities", Canadian Journal of Mathematics, 6(3), 393–404.

POSIX (2016). "The Open Group Base Specifications Issue 7", IEEE Std 1003.1-2008, 2016 Edition, http://pubs.opengroup.org/onlinepubs/9699919799/, accessed July 28, 2017.

A.H.G. RinnooyKan and G.T. Timmer (1987a). "Stochastic Global Optimization Methods Part I: Clustering Methods", Mathematical Programming 39, 27–56.

A.H.G. RinnooyKan and G.T. Timmer (1987b). "Stochastic Global Optimization Methods Part II: Multi Level Methods", Mathematical Programming 39, 57–78.

H.S. Ryoo and N.V. Sahinidis (1996). "A Branch-And-Reduce Approach to Global Optimization", Journal of Global Optimization, 8(2), 107–138.

M. Shafique (2017). "Heuristic Global Optimization", Ph.D. thesis, Systems and Computer Engineering, Carleton University, Ottawa, Canada.

L. Smith, J.W. Chinneck, and V. Aitken (2013a). "Constraint Consensus Concentration for Identifying Disjoint Feasible Regions in Nonlinear Programs", Optimization Methods and Software, 28(2), 339–363.

L. Smith, J.W. Chinneck, and V. Aitken (2013b). "Improved Constraint Consensus Methods for Seeking Feasibility in Nonlinear Programs", Computational Optimization and Applications, 54(3), 555–578.

Z. Ugray, L. Lasdon, J.C. Plummer, M. Bussieck (2009). "Dynamic Filters and Randomized Drivers for the Multi-Start Global Optimization Algorithm MSNLP", Optimization Methods and Software, 24(4–5), 635–656.

Z. Ugray, L. Lasdon, J.C. Plummer, F. Glover, J. Kelly, and R. Marti (2007). "Scatter Search and Local NLP Solvers: a Multistart Framework for Global Optimization", INFORMS Journal on Computing, 19(3), 328–340.

S.A. Vavasis (1995). "Complexity issues in global optimization: a survey", in Handbook of Global Optimization, 27–41, Springer.

A. Waechter and L. T. Biegler (2006). "On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming", Mathematical Programming, 106(1), 25–57.