Multistart Constraint Consensus for Seeking Feasibility in NLPs

by

Matthew R. MacLeod

A thesis submitted to
The Faculty of Graduate Studies and Research
in partial fulfilment
of the requirements for the degree of
Master of Applied Science in Electrical Engineering

Department of Systems and Computer Engineering

The Ottawa-Carleton Institute for Electrical and Computer Engineering

Carleton University
Ottawa, Ontario
September 8, 2006

Your file Votre référence
ISBN: 978-0-494-18326-7
Our file Notre référence
ISBN: 978-0-494-18326-7

Abstract

The Constraint Consensus (CC) method has empirically demonstrated a great deal of effectiveness at finding feasible problems to nonlinear programs. When used as a pre-solver for a local nonlinear solver, the effort expended by the solver drops dramatically with relatively little CC computation. The new Multistart CC (MCC) method is shown herein to be able to find feasible solutions to even more models than the strongest combination of single initial point selection, CC method, and local solver. Effective parameters are established for using CC on larger models than those previously examined, and an effective reduced starting search area is defined.

Acknowledgements

Although mine is the name on the front of this document, obviously the work of many contributed to its creation. First and foremost I'd like to thank my supervisor, Dr. John Chinneck, for invaluable patience and advice, endless proofreading and analysis, for contributing a few of the diagrams, and of course for devising the Constraint Consensus method upon which this work builds.

In my seven years at Carleton I've of course had many other professors and supervisors who have contributed in some way to my success, specifically Dr. Trevor Pearce, who's been a teacher, supervisor and mentor more times than I can count.

Of course no one completes any sort of engineering degree without some help in innumerable ways from fellow students. Andrew Preston, Laurence Smith, Mahdi Javer and Megan Howell Jones have all contributed in direct and indirect ways to this document, whether it be partnering on assignments, figuring out dates and regulations, discovering corner cases in LaTeX and MATLAB, or very occasionally encouraging me to work.

Lastly, a huge thanks must go to one Shauna Mullally, who although she made an appearance at the beginning of these seven years, waited until I started writing this paper to become an essential part of me. She is present in every word.

The undersigned recommends to

The Faculty of Graduate Studies and Research

acceptance of the thesis

Multistart Constraint Consensus

for Seeking Feasibility in NLPs

submitted by

Matthew R. MacLeod, B.Eng.

in partial fulfilment

of the requirements for the degree of

Master of Applied Science in Electrical Engineering

Thesis Supervisor

Chairperson, Department of Systems and Computer Engineering

Carleton University

September 8, 2006

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The need to optimize a given function subject to a series of constraints is a problem that crosses the bounds of engineering, economics, and manufacturing. Optimization problems (or *programs*, as they are traditionally known) are in general well understood and easily solved if the function and its constraints are all linear. Finding the global optimum of a nonlinear objective function, however, is typically much more difficult. Even finding a *feasible solution* (a set of variable values that satisfies all the constraints) can be difficult if one or more of the constraints is nonlinear, and a feasible solution may be enough to satisfy the creator of the problem. This is particularly true for *constraint satisfaction* problems, which are essentially optimization problems with no objective function. Rapid discovery of feasible solutions to *nonlinear programs* (NLPs) is the objective of the algorithms presented herein.

This work builds on the previous development of the *Constraint Consensus* (CC) method [6], [15]. As a *point improvement* heuristic, the objective of CC is to quickly and cheaply improve a point before passing it to a nonlinear solver. Tests of several different variants have shown it to be effective at both improving the probability that

several different solvers will reach feasibility from a given point (which will be referred to as the *success fraction* throughout the paper), and in reducing the average number of iterations required by a solver to complete. As the probability of success and time to complete are the two most practical performance metrics of a nonlinear solver, CC has demonstrated its effectiveness in improving points.

A closely related problem to point improvement is that of *initial point selection.* CC allows us to improve a given point once generated, but many initial point generation methods exist. Common choices include a random point, the origin, and the middle or edge of the bounds. The previous results [15] showed that the best point placement method combined with CC achieved success fractions in the 90% range, when paired with the stronger nonlinear solvers. However, no combination of methods was able to reach feasibility for all of the problems. It is not reasonable to assume that there exists a combination of point selection, improvement method and solver capable of succeeding 100% of the time from a single point, so the problem becomes that of how to select multiple initial points.

NLP methods that use multiple starting points are referred to as *multistart methods.* If a solver tries new points one at a time, it is clear that a higher probability of success per point will lower the average number of points that need to be tested before feasibility is reached. Further, if the average time per point is lower, the total time to evaluate a given number of points will be reduced. As CC makes improvements in both these areas when paired with a nonlinear solver, integrating it into a multistart framework should reduce the total time to find a feasible solution.

Results of the research will be presented in two stages. First, the performance characteristics of CC will be explored on larger models than those used in the previously published research, and good parameter settings determined. It will then be

shown that given a difficult test set of NLPs Multistart Constraint Consensus (MCC) methods can find feasible solutions to problems that previous single start methods can not, and that MCC methods are more efficient than a random search.

In addition, several ancillary contributions are included as part of the MCC research. It will be shown that a relatively small area (on the order of 1×10^4) contains a feasible point for an overwhelming majority (95.5%) of models in a large and varied test set, and as such any feasibility seeking algorithm should begin by searching this area. In regards to CC performance, means of identifying the point of diminishing returns in computational effort are presented, both when it is used on its own and in combination with a local nonlinear solver.

Chapter 2

Background on NLP

Although it is simple to describe a constrained optimization problem as finding the best value of a function subject to a series of constraints, it is useful to state the formulation of a nonlinear program more formally for clarity. Specifically, one would like to:

$$\min_X f(X) \text{ or } \max_X f(X) \tag{2.1}$$

where X is a vector of n variables, subject to m constraints of the form:

$$g(X) \{\leq, \geq, =\} b \tag{2.2}$$

and each variable is subject to upper and lower bounds:

$$l_i \leq x_i \leq u_i, \quad i = 1 \ldots n \tag{2.3}$$

It is easy to see that minimization and maximization are equivalent, i.e.:

$$\min_X f(X) \equiv -\max_X f(X) \tag{2.4}$$

and so we will refer to finding the optimum of an NLP where applicable, rather than discussing minimization and maximization.

Further, it was stated in the introduction that the primary goal of MCC is to find feasible solutions, rather than optimal solutions. However, feasibility seeking algorithms can also be used to seek optimality. The simplest conversion is as follows:

1. Find a feasible solution, and let its associated objective function value be v
2. Add $f(X) \geq v + \Delta$ (for maximization) to the set of constraints, where Δ is some small tolerance
3. Go to Step 1

One must of course be careful to specify reasonable stopping conditions, or this process may continue indefinitely after the global optimum is found.

2.1 Difficulties in Finding Feasibility for NLPs

As alluded to above, in comparison to linear programs (LPs) it is much harder to find feasible solutions for NLPs. Comparing the two types of problems provides a good introduction to the difficulties involved with NLPs.

1. *Feasible regions* - an LP has at most one feasible region. This is easiest to visualize in the two dimensional case; given a series of linear inequalities, these

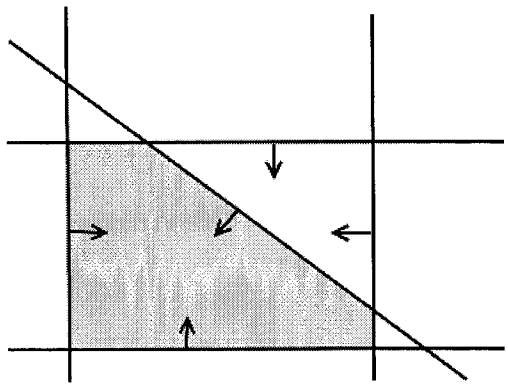

Figure 2.1: LP With a Feasible Region

lines either define the edges of a feasible polytope (see Figure 2.1), or there is no feasible region. If some set of them conflict such that there is no feasible solution (i.e. form an *infeasible set*), algorithms exist to detect this condition reliably and quickly. Constraints in the nonlinear case, however, may be arbitrarily curved. This means that instead of zero or one feasible regions, there are potentially many. See for example Figure 2.2, depicting the constraints:

$$y \leq \sin(x)$$

$$y \geq \cos(x)$$

It is easy to see that there are in fact infinitely many feasible regions created by this simple set of constraints.

2. *Multiple Optima* - due to the structure of an LP, there is either one solution (at the intersection or *corner point* of two or more constraints bounding the feasible region) or infinitely many solutions given by a linear function (connecting two or more corner points). As an NLP may have more than one feasible region,

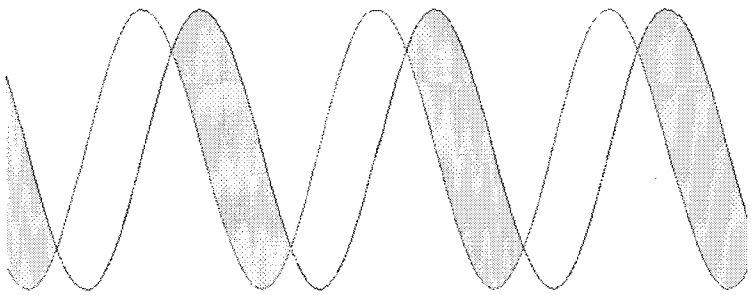

Figure 2.2: Infinite Feasible Regions Created by Trignometric Inequalities

this implies that each may have its own local optimum value(s). Further, as the objective function itself may be nonlinear, it is likely to have many local optima (see Figure 2.3). However, traditional analytical methods for verifying the optimality of a given point (e.g. the Karush-Kuhn-Tucker conditions [17, 18]) generally apply only in the local sense (i.e. they cannot guarantee that a given point is globally optimal). Verifying that a given local optimum is in fact the global optimizer is a much more difficult problem in an NLP, and involves a great deal of computational effort. As such, heuristic methods for finding good (but possibly sub-optimal) solutions to NLPs are very popular, especially if they have the capability to return many solutions. This problem of local optima also afflicts feasiblity seeking algorithms, as they may become trapped at points that locally minimize some measure of feasibility violation.

3. *Convexity* - it has been said that "the great watershed in optimization isn't between linearity and nonlinearity, but convexity and nonconvexity" [31]. Given a feasible convex set of constraints and a convex objective function, it can be shown that there is one feasible solution that is globally optimal. Several

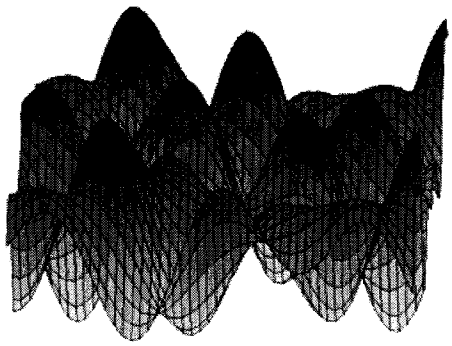

Figure 2.3: Multiple local optima in:
$f(x,y) = -0.2(sin(x+4y) - 2cos(2x+3y) - 3sin(2x-y) + 4cos(x-2y))$

methods (e.g. interior point methods) have been developed that can exploit this property to efficiently solve NLPs in this class. Although these are highly practical in many situations, two concerns arise:

(a) constraints of greater than second order are unlikely to be convex, and

(b) those who need to solve optimization problems are not necessarily mathematical experts, and automated tools for identifying convex sets are still imperfect at best.

Other methods exist which rely on properties other than convexity holding true for the objective and/or the constraints (e.g. Lipschitz continuity, bilinearity/biconvexity), but similar concerns arise. An ideal, generic NLP solver should not require deep mathematical knowledge on the part of the user.

4. *Complicated Functions* - the class of nonlinear functions includes the most memory intensive, difficult to evaluate piecewise polynomials and trigonometric functions. In the extreme, it may be a simulation or other "black box" function (e.g.

simulation of an aircraft) that takes days or more to evaluate. When working with a black box, one does not have access to the derivatives and other analytical information of the objective function on which many traditional methods rely. In these cases a direct search method must be used, relying only on the value of the objective function or constraint to make decisions on what points to explore. Generating promising feasible points is often the primary goal of the modeller under these conditions.

5. *Misleading constraints* - given the variety of function shapes and properties encompassed by nonlinear functions, NLP solution methods usually have at least one general situation in which they perform poorly. In the case of feasibility seeking algorithms, the worst situation often occurs when dealing with a region in which two constraints almost (but do not quite) touch. As linear constraints do not "bend" toward or away from each other, this situation is not possible in an LP. An example of such constraints is shown in Figure 2.4. As an algorithm approaches point P it will find the sum of infeasibilities decreasing, as it is very close to satisfying both constraints. It can be difficult for an algorithm to identify reliably that it is actually impossible to reach feasibility in this region. Essentially, it is a local optimum (in the feasibility sense) at which an algorithm is likely to terminate.

Given these properties of NLPs, it is not particularly surprising that no one method has been developed that is clearly superior on all problem classes. Certain methods work well on certain types of problems, but again it may be difficult for an inexperienced modeller to know *a priori* which one to select.

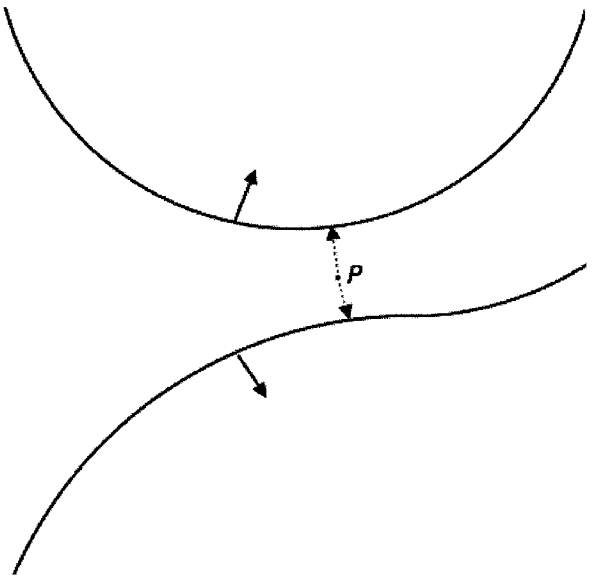

Figure 2.4: Misleading Constraint Curvature

2.2 Measuring Closeness to Feasibility

When searching for feasibility rather than optimality, one is interested in satisfying the constraints rather than optimizing the objective (if any). It is worthwhile to define the most common metrics used to measure the feasibility violation at any given point:

SINF - the *Sum of the Infeasibilities* is most commonly defined as the sum of all the constraint violations. Attempts to reach feasibility by minimizing this value can be greatly affected by constraint scaling, with the effect of some poorly scaled constraints dominating, while others have no effect. To mitigate this effect, some methods use an altered definition of the metric. Care must therefore be taken when comparing SINF values across methods.

NINF - the *Number of Infeasibilities* is the number of constraints violated at a given point.

Penalty Functions - are usually defined as weighted sums of the constraint violations (or squared violations), and are designed to monotonically decrease as points move closer to a feasible region.

A further property of interest is the shortest vector from a given infeasible point to a given violated constraint, as the length of this vector gives a lower limit on the Euclidean distance to feasibility, or *feasibility distance.* If this is the only constraint, it also gives the direction to move in order to satisfy the model. Adding this *feasibility vector* to the current point gives the *orthogonal projection* of the point onto the constraint. *Projection algorithms* originated in [7], and have recently been developed further by Censor in collaboration with several others (see [2, 4, 3, 38]). Constraint Consensus (see Section 3.3.3) is one such method, which uses combinations of the feasibility vectors for all of the violated constraints to seek feasibility.

It is important to note that when most nonlinear solvers discuss *constraint violation* and *feasibility tolerance* they are referring not to the distance to feasibility in the variable space, as above, but the function space of the constraints. Taking the inequality constraint $LHS \leq RHS$ as an example (i.e. left hand side $\leq$ right hand side), it is considered satisfied if $LHS \leq RHS + \epsilon$, where ϵ is some feasibility tolerance (usually on the order of 1×10^{-6}). It can be seen that the tolerance is measured against changes in the constraint function, rather than in the variables themselves.

The danger of using a function space measure of "closeness" to feasibility, is that it is sensitive to the scaling of the constraints. Multiplying every term in the constraint by a very small number can make the feasibility tolerance trivially satisfiable, whereas multiplying by a large number exaggerates the violation for the same variable values. Using a variable space measure avoids this issue.

Chapter 3

Multistart and Reaching Feasibility—The State of the Art

Having summarized the basic strategies of existing optimization methods for NLPs, it is worth examining the current state of the art in the areas of interest—multistart, initial point placement and feasibility seeking. Two recent developments of specific relation to the research at hand are the *Multistart NLP* [20, 21] method of Lasdon and Plummer and the *Constraint Consensus* method of Chinneck and Ibrahim [6, 15]. These papers present feasibility seeking results, the former with a related multistart method for continuous NLPs, and the latter with an extremely efficient singlestart method. Both also deal with improved starting point heuristics as part of their methodology. Their context in the larger state of the art will be explored in the following sections.

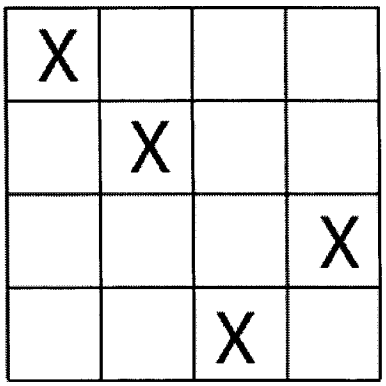

Figure 3.1: Two-Dimensional Latin Hypercube Sample [32]

3.1 Initial Point Placement

3.1.1 Constrained Random

A purely uniform random search of the variable space in a problem of several hundred dimensions intuitively seems rather hopeless [28, Exact GO Approaches]. This is particularly true if the specified bounds are rather wide (or if the variables are not bounded at all).

In order to reduce some of the variance of traditional random Monte Carlo simulation, the *Latin Hypercube Sampling* statistical method [23] can be used to more evenly distribute points in space (thereby requiring fewer points to achieve the same reduction in uncertainty). To produce M points in a latin hypercube sample, each axis is first divided into M equally sized segments. One random coordinate is then selected within each segment for each axis. Finally, these coordinates are randomly combined to form M points. Picturing the two dimensional case, the result is an evenly spaced grid in which there is exactly one point within each row and within each column (see Figure 3.1). Jones *et al.* used this method in order to select good ini-

tial points for their stochastic modelling approach to optimizing black box functions [16], as it provides a very even knowledge of the search space.

3.1.2 Bounded Start

The difficulty of using random or guided random point generators in an unbounded space is highlighted in [21]. Often a model with "unbounded" variables has been insufficiently bounded by an inexperienced or lazy modeller, and exploring much of the space leads to numerical errors, infeasible results, or slow progress. The authors found that by artificially restricting variables with very large or no bounds to $\pm 1 \times 10^4$ or $\pm 1 \times 10^2$ increased the likelihood that a given random solver launch would return without error, but sometimes excluded the area of space containing the global optimum. They determined that the correct value is problem specific, and emphasized good bound selection on the part of the modeller.

3.1.3 Improved Initial Point Placement Heuristic

When looking to place a starting point for a solver, the simplest choices are either a random point or the origin. As the origin may not even be within the variable bounds, a commonly applied heuristic (see e.g. [26]) sets the initial coordinates as follows:

- if the variable is bounded above and below: set at the midpoint,
- if the variable is singly bounded: place on the bound,
- if the variable is unbounded, place at zero.

Although this will tend to place the point relatively centrally, it can lead to numerical problems. Many coordinates will be set to zero, which is a problem for constraints with terms of the form $1/x$. Additionally, many coordinates are likely to be the same (either zero or due to a group of constraints with the same bound(s)), which causes a problem for terms of the form $1/(x_1 - x_2)$.

To mitigate these issues, Chinneck and Ibrahim [15] proposed the addition of a small randomized perturbation to the above heuristic. The resulting heuristic is as follows:

- if the variable is bounded above and below: set at the midpoint + Δ,
- if the variable is singly bounded below: place on the bound + Δ,
- if the variable is singly bounded above: place on the bound - Δ,
- if the variable is unbounded, place at zero + Δ.

Where Δ is a uniformly distributed random number between 0 and 1 (or smaller if the variable bounds define a smaller range). Empirical results presented in the paper found that with few exceptions, the percentage of problems for which solvers were able to reach feasibility was higher when CC was combined with this heuristic than with the standard version, the origin, or a random point.

3.2 Multistart Methods for Global Optimization

The solution methods discussed thus far have all been concerned with providing a single initial point. Two main problems exist with this approach when using a feasibility seeking algorithm:

- Initialize $i = 1$

1. Repeat until stopping condition is met:
   - 1.1 Generate: Construct solution x_i
   - 1.2 Search: Apply a search method to improve x_i. Let x_i' be the solution obtained.
   - 1.3 If (x_i' improves the best solution) update the best solution
   - 1.4 $i \leftarrow i + 1$

Figure 3.2: Multi-Start Procedure [22]

- the algorithm may terminate unsuccessfully at an infeasible point, and
- given a poor choice of initial point, the algorithm may proceed very slowly.

As such, it is often desirable to start from many different points in the space. In this case, the method for selecting points to try becomes a major issue, as well as the termination conditions of individual runs. The termination conditions for the algorithm as a whole are quite important in the optimization case (as there is no way to analytically determine if a given local optimum is the global optimum). In the feasibility seeking case, however, merely locating a feasible solution is an easily defined stopping condition (with time limits serving as a check on runaway invocations).

A good summary of current multistart methods for combinatorial optimization is given in [22]. The basic "Multi-Start" procedure described therein is equally applicable to continuous feasibility problems, and is reproduced in Figure 3.2. The simplest and most naive possible class of multistart methods uses a purely random generator for the 'generate' step, and applies a local solver as the 'search' step. Some specific examples will be discussed below.

3.2.1 Combinatorial Methods

One common element of multistart methods involves generating a pool of good solutions, and then somehow mixing them together to create potentially better solutions. Due to their structure, they are especially popular for problems with a finite number of discrete solutions. The general idea, however, is applicable to NLPs as well.

The most obvious example is *Genetic* (or *Evolutionary*) *Algorithms* (see e.g. [14]). A pool of trial solutions is created, which are then combined and/or 'mutated' in a rough analogy to species evolution. At each stage the best new solutions replace the worst in the pool in a 'survival of the fittest' manner. In more advanced versions some relatively poor solutions are kept in the pool to maintain diversity and prevent the search from becoming too localized. Many variations have been proposed, and this is still a very active area of research.

A *GRASP* (see e.g. [29]), or Greedy Randomized Adaptive Search Procedure, is a method that keeps a restricted list of the best known candidate solution components, and randomly selects among them to build a possible solution. For example, consider a set-covering problem, wherein the candidate list would consist of possible subsets to include in the solution. A purely greedy choice would always choose to add the subset that covered the most currently uncovered items. However, the candidate list in a GRASP set covering implementation would include, for instance, the top 10% of greediest choices, or the top five choices (these values are open to parameter tuning). A random choice is made amongst the candidate list at each step. This is only one example of its application, and the concept can be applied in many different settings. Local searches are generally performed around candidate solutions, as in most multistart methods.

GRASP has also been integrated with the *Path Relinking* framework [30], devel-

oped originally in association with the *Scatter Search* method [12]. In Scatter Search, clusters of points in different regions of the problem space are identified, and are combined to form weighted centres of the regions. Linear combinations of subsets of the points are combined both within and across regions in an attempt to find better solutions. The path relinking aspect involves exploring trajectories between good solutions, on the theory that even better solutions may be located in between. Elements of one solution are progressively added to the other, with optional local improvement searches every few steps. The theme of using the information from previously evaluated points is an important one in multistart methods, and MCC in particular.

3.2.2 Probabilistic Methods

As mentioned in Section 2.1, black box functions are often the most challenging to deal with in the realm of NLPs. The *EGO* (Efficient Global Optimization) method aims to sample a select few points of an expensive function, and fit a surface to them [16]. The function behaviour is modelled stochastically, and draws on previous tradition in mathematical geology (kriging), bayesian global optimization, and statistics. Successive samples are biased towards two kinds of areas—those that are strongly predicted to have better function values, and those where the uncertainty of the function is so large than a better function value is quite possible, regardless of the predicted value. One of the key insights is the use of a latin hypercube derived initial sample to start with a minimal amount of uncertainty. However, for best results, [16] recommends an initial sample with around 10 times as many points as the number of dimensions, which may be impractical for truly expensive functions.

3.2.3 Continuous Methods

As noted in [10], there are very few methods that treat general NLPs, and few have been tested for engineering applications. Most are developed for unconstrained problems, and then use some form of penalty method to deal with constraints (see Section 3.3.1). The paper defines a *clustering method* as one in which local search is (ideally) only used once per local optimum. Clusters of points in the same region of attraction (i.e. near the same optimum) are somehow identified so that only one point per cluster is used to initialize an expensive local search, although the assertion is made in [10] that local search performance tends to depend heavily on dimension. Most of these methods (see e.g. [35, 36]) require solving a series of quadratic sub-problems (i.e. NLPs) at each step to identify these clusters, which can be quite expensive. Nonetheless, *sequential quadratic programming* is one of the most effective methods available for general constrained NLPs, and is implemented by many commercial solvers—including KNITRO [37], which was found to be the most effective local solver evaluated in previous CC research [15].

The combinatorial methods above can also largely be applied to continuous problems with minor changes. However, they tend to require many function evaluations and local searches. For the reasons enumerated in Section 2.1, invoking a local nonlinear solver is expensive. As such, it is desirable to be especially vigilant in filtering points *before* they are passed to the solver, in order to avoid unnecessary invocations. An *acceptance-rejection* technique is a multistart method that has been modified to only perform local searches on points of a certain quality. In general, the filtering becomes more severe as the algorithm progresses. Extensions of this general approach are used in the Zooming and Domain Elimination methods of [10], MSNLP[21], and many other methods.

MSNLP

In addition to its use of acceptance-rejection ideas, MSNLP [21] incorporates a method of identifying the regions of attraction of the local minima. The *distance filter* compares each generated trial point to the set of best known solutions, and rejects if it is within a solution's estimated *basin of attraction* (analogous to the clusters in clustering methods). MSNLP approximates this region with a hypersphere, and obtains an estimate of its radius by using the distance from the starting point for each returned solution. This radius is dynamically increased if another starting point returns the same solution from a greater distance, and decreased if the radius around this solution has rejected too many trial points. Further extensions have initialized the radius to be slightly smaller than the calculated radius, and have also dynamically adjusted the basins so they do not overlap. The general idea is similar to the *repulsion algorithm* [34], except that with repulsion every new point is made use of by 'pushing it away' from the previously solved points in whose basins it lies. This is perhaps more computationally intensive than outright rejecting a point and moving on, depending on the problem characteristics.

The *merit filter*, similar to classic acceptance-rejection technique, compares the penalty function value of a point with a certain threshold. Whenever a point with the new lowest value of the penalty function is discovered, the threshold is set to this value. Like the merit filter, if the threshold has rejected too many points in a row (the default value is 20) it is relaxed by a certain percentage.

After a stage 1 in which n_1 points are generated and their penalty function values evaluated, stage 2 begins with the best point found in stage 1. This point is passed to the local solver, and the filters initialized with the result. In this stage n_2 points are generated one by one. If any one passes the two filters, it is passed to the solver in

order to find a new solution point. Using the resulting point the filters are updated and the next iteration is continued. It should be noted that these returned points may not actually be locally optimal (or even feasible). Iterations are halted when a certain target value of the objective function is obtained, or when progress has slowed.

3.3 Feasibility Seeking

There are two main approaches to feasibility seeking—converting the problem to an optimization problem, or dealing with feasibility as a separate issue. Both approaches will be discussed below.

3.3.1 Penalty Functions and Phase 1

Perhaps the most common method of dealing with constraints in constrained feasibility and optimization problems is through the use of penalty functions. Generally, some function is constructed that increases with increasing constraint violations, and potentially also decreases with increasing distance from satisfied inequality constraints. Minimizing steps on such a function will aim to satisfy violated constraints, while also ensuring that the satisfied constraints remain satisfied. These are often referred to as *Phase 1* methods, as some optimization methods require a feasible solution to be found (Phase 1) before they can seek optimality (Phase 2).

For example, the *Quadratic Penalty Method* described in [9], minimizes a penalty functions of two parts, defined as:

$$\Phi_1 = \frac{1}{2}\sum_{i=1}^{p}[g_i(X)]^2 + \frac{1}{2}\sum_{i=p+1}^{m}[g_i(X) + |g_i(X)|]^2 \tag{3.1}$$

Where $g_1(X) \ldots g_p(X)$ are equality constraints of the form $g_i(X) = 0$ and $g_{p+1}(X) \ldots g_m(X)$ are inequality constraints of the form $g_i(X) \leq 0$ (note that all of the constraints described in Equation (2.2) can be rearranged to one of these two forms, so there is no loss of generality). Satisfied constraints will produce a value of zero in either summation, and so minimization halts successfully if $\Phi_1 = 0$. If minimization halts at a local minimum $\Phi_1 \neq 0$, another starting point must be chosen.

3.3.2 MSNLP: Feasibility Seeking

In a modification to support feasibility seeking, the MSNLP algorithm (see Section 3.2.3) replaces the penalty function objective with the sum of the infeasibilities [20]. In this case the returned points used to form the basins of attraction are guaranteed to not be feasible, as finding a feasible point terminates the algorithm successfully.

A selection of their results in feasibility seeking mode is presented in Table 3.1 (organized by number of variables N and number of constraints M). The problems are reactor network design problems from the GlobalLib [11] set. Solutions satisfying a feasibility tolerance of 1×10^{-6} are found for all problems except ex8_3_7. For the solved problems, between 3 and 24 solver calls are required (Conopt, in this case). MSNLP limits on total iterations and stage 1 iterations were 1000 and 200, respectively. It is interesting to note that in all cases using a random point generator rather than the OptQuest (OQ) scatter search implementation [19] is more efficient (i.e. fewer solver calls are made).

Table 3.1: Finding Feasible Solutions to Reactor Network Design Problems Using MSNLP/Conopt [20]

Problem Name	N	M	Final SINF with OQ driver	Solver calls with OQ driver	Solver calls with random driver	MSNLP time with OQ driver
ex8_3_1_noinit	116	77	4.91×10^{-13}	3	2	0.72
ex8_3_2_noinit	111	77	1.27×10^{-07}	3	2	0.71
ex8_3_3_noinit	111	77	1.48×10^{-05}	3	2	0.67
ex8_3_4	111	77	9.68×10^{-08}	10	2	1.15
ex8_3_5	111	77	2.27×10^{-10}	6	2	1.21
ex8_3_6_noinit	111	77	3.03×10^{-12}	20	2	1.55
ex8_3_7	127	93	$1.00 \times 10^{+02}$	65	2	4.15
ex8_3_8	127	94	6.38×10^{-13}	24	2	2.67
ex8_3_9_noinit	79	46	4.67×10^{-12}	11	2	1.64
ex8_3_10	142	109	1.47×10^{-07}	16	3	1.66
ex8_3_11_noinit	116	77	1.99×10^{-06}	3	2	1.14
ex8_3_12_noinit	121	82	3.43×10^{-09}	4	2	1.81
ex8_3_13_noinit	116	73	1.49×10^{-12}	4	2	1.19
ex8_3_14	111	72	2.11×10^{-11}	4	3	1.07
totals				176	30	10.32

3.3.3 Constraint Consensus

As experience has shown that the success rate of a nonlinear solver is highly dependent on the provided initial point, the CC algorithm was developed as a computationally inexpensive method of improving points before they are passed to a local solver [6]. The algorithm achieves this by attempting to move closer to feasibility at each iteration, using the gradient information of the violated constraints. As part of this research, several different initial point placement methods were evaluated, and a new and successful one developed.

It is useful to first consider the case of satisfying a single constraint, and how to move from an infeasible point to a feasible one. To move to the nearest feasible point, one must calculate the orthogonal projection of the infeasible point to the violated constraint. In CC terms, the vector extending from the infeasible point to the closest feasible point is defined as the feasibility vector. The feasibility vector for a constraint i is calculated as:

$$fv_i = \frac{v_i d_i \nabla c_i(C)}{\|\nabla c_i(C)\|^2} \tag{3.2}$$

where:

- $\nabla c_i(C)$ is the constraint gradient,
- $\|\nabla c_i(C)\|$ is its length,
- v_i is the magnitude of the constraint violation $|c_i(X) - b_i|$, or 0 for satisfied constraints,
- d_i is $+1$ if $c_i(X)$ must be increased to satisfy the constraint, and -1 if it must be decreased.

The vector is exact for linear constraints, but equation (3.2) constitutes only a linear approximation for a nonlinear constraint.

For a problem with multiple violated constraints, the question of how to combine multiple feasibility vectors then arises. In CC, each violated constraint that contains a given variable is used in determining how far (and in which direction) to move in that dimension. In the original algorithm (see Figure 3.3), the relevant component of each eligible feasibility vector is averaged. By combining the results for each dimension, a *consensus vector* is formed. This consensus vector is added to the current point, and the process repeated until the stopping conditions are met.

The algorithm terminates successfully if the magnitude of each feasibility vector is less than a feasibility distance tolerance α. Progress is deemed to have halted and a run deemed unsuccessful if the magnitude of the consensus vector falls below a movement tolerance β, or if a number of iterations μ is exceeded. Although not shown in Figure 3.3, if at any point a constraint or gradient evaluation results in a numerical error, that constraint is ignored for the purpose of calculating the consensus vector at the current iteration. However a flag is raised, and if the stopping conditions are otherwise met on that iteration, the run is considered unsuccessful.

Creating the Consensus Vector: Algorithmic Variations

Naturally, averaging the components of the feasibility vectors is only one method of arriving at a consensus vector. Two main categories of methods for doing so were explored in [15]—*feasibility distance* based and *direction based.* Feasibility distance based versions (FDnear and FDfar) select either the shortest or the longest feasibility vector as the basis for movement (with the other components not in the vector obtained by averaging, as before). In the direction based versions, first the most

Inputs:

- a set of I constraints $c_1 \ldots c_i$ in J variables $c_1 \ldots c_j$,
- an initial point x,
- a feasibility distance tolerance α,
- a movement tolerance β,
- a maximum number of iterations μ,

1. Repeat μ times:
   - 1.1 $NINF = 0$; for all $j : n_j = 0, s_j = 0$.
   - 1.2 For every constraint c_i :
     - 1.2.1 If c_i is violated (without considering a tolerance) then:
       - 1.2.1.1 Find the feasibility vector and the feasibility distance.
       - 1.2.1.2 If $\|fv_i\| > \alpha$ then:
         - $NINF = NINF + 1$
         - For every variable x_j in c_j: $n_j \leftarrow n_j + 1; s_j \leftarrow s_j + fv_{ij}$
   - 1.3 If $NINF = 0$ then exit successfully.
   - 1.4 For every variable x_j:
     - 1.4.1 If $n_j \neq 0$ then $t_j = s_j/n_j$, else $t_j = 0$.
   - 1.5 If $\|t\| \leq \beta$ then exit unsuccessfully.
   - 1.6 $x \leftarrow x + t$
   - 1.7 If necessary, reset x to respect any violated variable bounds.
2. Exit unsuccessfully.

Figure 3.3: The Basic Constraint Consensus Algorithm [6][15]

"popular" direction is selected for each dimension, and then either the maximum, minimum, or average movement is made in that direction.

It is useful to describe the direction based averaging (DBavg) version (Figure 3.4) of the algorithm in detail, as an example of the possibilities. A count of the number of violated constraints (n_j^+ and n_j^-) and the total magnitude of suggested movement (s_j^+ and s_j^-) is kept in each direction for each dimension. If for a given dimension more constraints suggest movement in either the positive or negative direction, an average of the movements in that direction is taken. If the number of violated constraints in each direction is equal and not zero, all suggested movements are averaged (as in the original algorithm).

Also of note in Figure 3.4 is that a new progress parameter $prog_{max}$ has replaced the movement tolerance β. Further unpublished research by Chinneck has discovered that unless the magnitude of the consensus vector is shrinking at each consecutive iteration, the algorithm is not likely making useful progress. To make use of this insight, $prog_{max}$ can be used as a *progress tolerance* to terminate unproductive CC invocations. It defines the maximum acceptable *progress ratio* of $\|t^k\|/\|t^{k-1}\|$, where t^k is the consensus vector at iteration k. It was found that this covered many of the cases where the movement tolerance was triggered, which was already very rare in problems of any significant size. As such, β was removed for simplicity.

Relevant Results: Single Start Constraint Consensus

In almost all observed cases, application of CC both increases the success fraction of a solver, and reduces the amount of effort the solver expends to reach success (i.e. a feasible solution). This subsection presents some of the relevant supporting data, as published in [15]. All results are based on tests with 231 models from the CUTE

Inputs:

- a set of I constraints $c_1 \ldots c_i$ in J variables $c_1 \ldots c_j$,
- an initial point x,
- a feasibility distance tolerance α,
- a maximum number of iterations μ,
- a progress tolerance $prog_{max}$

1. For each iteration k, $k = 1 \ldots \mu$:
   - 1.1 $NINF = 0$; for all $j : n_j^+ = 0, n_j^- = 0, s_j^+ = 0, s_j^- = 0$.
   - 1.2 For every constraint c_i :
     - 1.2.1 If c_i is violated (without considering a tolerance) then:
       - 1.2.1.1 Find the feasibility vector and the feasibility distance.
       - 1.2.1.2 If $\|fv_i\| > \alpha$ then:
         - $NINF = NINF + 1$
         - For every variable x_j in c_j:
           If $fv_{ij} > 0$ then $s_j^+ = s_j^+ + fv_{ij}, n_j^+ = n_j^+ + 1$
           If $fv_{ij} < 0$ then $s_j^- = s_j^- + fv_{ij}, n_j^- = n_j^- + 1$
   - 1.3 If $NINF = 0$ then exit successfully.
   - 1.4 For every variable x_j:
     - 1.4.1 If $n_j^+ = n_j^-$ and $(n_j^+ + n_j^-) > 0$ then $t_j^k = (s_j^+ + s_j^-)/(n_j^+ + n_j^-)$
     - 1.4.2 Elseif $n_j^+ > n_j^-$ then $t_j^k = s_j^+/n_j^+$
     - 1.4.3 Else $t_j^k = s_j^-/n_j^-$
   - 1.5 If $k > 1$:
     - If $\|t^k\|/\|t^{k-1}\| > prog_{max}$ then exit unsuccessfully.
   - 1.6 $x \leftarrow x + t^k$
   - 1.7 If necessary, reset x to respect any violated variable bounds.
2. Exit unsuccessfully.

Figure 3.4: Average Direction-Based (DBavg) CC with Progress Tolerance

set [13], namely those with fewer than 300 constraints and variables, and at least one nonlinear constraint.

Table 3.2 shows the success rate of the CC methods above when paired with the KNITRO solver [37], and launched from various initial points. The points evaluated are a uniformly distributed random point, the origin, the standard heuristic point described above, the randomized version of the heuristic described in Section 3.1.3, and the point provided in the model. Where a point was not provided in the model (14% of the models), the point was set to the origin (hence the label CUTE/Origin in the tables).

Table 3.2: Solver Success Fractions

CC	Random	Origin	Std. Heur.	Ran. Heur.	CUTE/Origin
None	0.714	0.749	0.749	0.900	0.931
Basic	0.745	0.775	0.766	0.909	0.913
DBAvg	0.771	0.771	0.758	0.905	0.918

It is interesting to note that applying CC improves the success fraction in all cases except when the model contains a modeller supplied point. Of the five solvers tested, this effect only occurred with KNITRO. It is conjectured that because KNITRO uses a barrier method, the tendency of CC to place points near the limiting values of the constraints may be problematic. As the multistart research is mainly interested in cases where a good point is *not* provided, however, this is not a significant negative.

It can also be seen in Table 3.3 that applying CC significantly reduces the average number of solver iterations required to find a solution. In the case of a random point, it does so by more than half. As multistart with a reasonable number of points requires some form of random or directed random points, this is the most significant case. It is also important to note that the time consumed by the CC iterations is

very low compared to that of the nonlinear solvers. The ratio of average solver time to average CC time was found to vary from 9.1 to 2209.8 [15].

Table 3.3: Solver Iterations over Compared Subsets

CC	Random	Origin	Std. Heur.	Ran. Heur.	CUTE/Origin
None	24.4	17.9	17.7	15.0	17.1
Basic	10.4	11.7	14.2	14.3	12.9
DBAvg	11.2	11.3	13.7	16.9	14.0

As the success fraction numbers in Table 3.2 are calculated over the whole set, it does not make clear whether the methods that solve a greater number of problems solve all of the same problems as those that solve fewer (i.e. whether the weaker methods solve only a subset of those models solved by the stronger methods, or some different set). Table 3.4 shows the net effect of using the recommended CC methods with KNITRO for the different initial points. The "CC+" column is the number of models solved with CC but not without, the "CC-" column the opposite, and the final column the net change. Again the only net decrease is with the modeller provided points, although there are some losses with the other points that are compensated by larger gains. In general, it is more likely that one will get a better result by applying CC to a random point than not. This is therefore the recommended course of action.

Table 3.4: Net Success Using Constraint Consensus

initial pt	CC	With CC+	With CC-	Net Change With CC
random	DBavg	22	9	13
randomized std	FDfar	7	4	3
CUTE/Origin	DBavg	2	5	-3

In summary, CC has demonstrated much potential as a point improvement heuris-

tic. Even when paired with the strongest solvers, it can reduce the effort necessary to reach feasibility.

Chapter 4

Using Multistart Constraint Consensus to Reach Feasibility for Large Nonlinear Programs

Given both the success of Constraint Consensus as a singlestart presolver and the current research into multistart methods for dealing with generalized nonlinear programs, a natural area to explore is that of using CC to make multistart methods more successful and efficient. The question this thesis addresses is whether CC can be used as part of a multistart method to increase the effectiveness and efficiency with which feasibility can be reached for difficult NLPs. Multistart Constraint Consensus (MCC) considers the identification of a feasible solution to a nonlinear program as a successful run, and therefore this will be the criterion used to measure increases in effectiveness. Efficiency will be measured by the total amount of time used to solve a problem, as well as the number of times a nonlinear solver is invoked on that problem. The empirical evidence presented in the following sections will establish that

MCC can make improvements in both these areas, and is the key contribution of this research.

The CC method as a whole is relatively new, and all published work on the subject has been carried out by Dr. Chinneck and his current and former students. To the best of our knowledge, this is the first attempt to incorporate CC into a multi-start methodology, and the results obtained are unique. The most similar method of generating and filtering points for feasibility seeking is the MSNLP work of Lasdon and Plummer [21], but they have made several different design choices. The differences are largely due to the differences between feasibility seeking and optimization (for which their algorithm was originally intended), as well as the use of the additional information CC invocations provide. MCC also aims to use less complicated calculations than those in MSNLP.

Although in general feasibility seeking algorithms can be converted to seek optimality, and optimization algorithms can be converted to seek feasibility, there are some basic differences in approach. The original MSNLP algorithm first finds feasible local solutions, and uses these to bias further sampling. Infeasible solver returns are treated as a problem to be worked around. By contrast, MCC is interested in using the information from infeasible points to bias further sampling, and the identification of any feasible solution is the primary stopping condition. Whereas MCC is interested in locating probable feasible regions and sampling toward them, MSNLP expects to find locally optimal feasible solutions, and then discourages sampling around them. Due to the difference in approach, MCC is interested in a different class of problems than MSNLP, those in which merely reaching feasibility is expected to be very difficult. The results presented in [21] use problems from the GAMS Globallib that were present in the selection pool for the MCC research, but few met the difficulty screen-

ing criteria. The results presented for the feasibility seeking version of MSNLP in [20] also show a relatively large amount of effort and high number of solver launchers for obtaining feasible solutions. One of the primary objectives of MCC is to use simpler calculations and therefore expend less effort.

One of the primary advantages of CC in the single start case is that not only is it inexpensive and quick, but it requires no special prior knowledge of the model. This is especially important as increasing numbers of non-specialists (e.g. engineers, scientists, economists) create optimization problems as part of their day to day work. As a naive modeller who has purchased or downloaded a solver cannot be expected to know what a good initial point is (or what good parameter choices might be), it is important to provide a method of choosing new points that requires no user intervention or analysis. Providing a straightforward multistart method for such a modeller that is quick and effective is the primary focus of Multistart Constraint Consensus.

Chapter 5

Constraint Consensus and Large NLPs: Design Issues

The previous published research on Constraint Consensus [6, 15] evaluated the method using versions of the solver software limited to 300 variables and 300 objectives and/or constraints. Before applying CC in higher dimensions, it is necessary to first establish its performance characteristics in this larger space. Also, the results in [15] used a fixed value of α when pairing it with the various solvers, with the rationale that "it is worth spending a small amount of extra effort during the constraint consensus phase (which is computationally cheap) to provide the nonlinear solver with a starting point that is closer to feasibility." As the computational effort required to perform a CC iteration becomes non-trivial when dealing with many hundreds or thousands of constraints, the trade-off between CC and solver effort must be re-evaluated. Results for the experiments described here will be presented in Chapter 7.

5.1 Constraint Consensus for Exploring Space

The results presented in [15] show that it is more difficult for CC to reach feasibility from a random point than any of the selected 'special' points, whether on its own or paired with a solver. Initial experiences with larger models suggested this problem becomes even worse in higher dimensions, so a more detailed look was taken at the problem. Especially when applied to fully or largely linear models, the estimated distance to feasibility (as encapsulated by the magnitude of the consensus vector) at each CC iteration shows a tendency to rapidly plateau. The progress tolerance $prog_{max}$ introduced in Section 3.3.3 can be used to detect the 'knee' in the curve where this plateau begins. A primary concern is that if the initial point is too far away from the feasible region, this plateau may be reached before a reasonable α tolerance can be met.

5.2 Establishing a Good Starting Box

Unsurprisingly, models with unbounded variables (or very large bounds) cause the most problems for a feasibility seeking algorithm when starting from an arbitrary or random point. At great distances from a feasible region, estimates of nonlinear constraints are less reliable, functions may become undefined or numerically problematic, and many misleading local optima (in the sense of locally minimal but positive infeasibility measures) may exist en route to the true feasible region. Therefore it is quite desirable to identify a smaller region where feasibility is more likely to be located. As a demonstration, performance of the MSNLP algorithm for both feasibility seeking [20] and optimization [21] has been shown to improve by artificially limiting variable bounds to $\pm 1 \times 10^2$ and $\pm 1 \times 10^4$. A more thorough study of the issue is very relevant.

Intuitively, it seems natural that feasible regions are located somewhere towards the centre of a unbounded problem. A naive or inexperienced modeller may assume that a solver will start searching from a 'reasonably' low value, perhaps the origin, and will only move to much higher values if the problem warrants. Further, if a variable is singly bounded, it seems likely that the bound was placed because the modeller considered that the optimization process would tend to move past that value. A bound of this sort indicates that this is somehow an 'interesting' area to explore. This intuitive justification is expressed in the placement of the standard heuristic point at the edge of singly bounded dimensions, and at the centre of dimensions with no or both bounds.

Given that the standard heuristic (and especially its randomized version) have shown a very high probability of success when used as a starting point for various solvers, it seems reasonable to primarily search in the immediate area for further potential initial points. The question then becomes one of how large to make this area.

To help answer this question, one can find the closest feasible point x to the standard heuristic point y for any given model by finding:

$$\min_X f(X) = (x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2 \tag{5.1}$$

subject to all of the constraints of the original model. By examining the magnitude of the differences in coordinate values between x and y for a large number of problems, the appropriate magnitude Θ of the initial search area can be established. By analogy to the standard heuristic, this area will be defined as:

- If a variable is not bounded, $(-\Theta, \Theta)$

- If there is a single lower bound L, $(L, L + 2\Theta)$
- If there is a single upper bound U, $(U - 2\Theta, U)$
- If there are two bounds separated by $> 2\Theta$ with centre C, $(C - \Theta, C + \Theta)$
- If there are two bounds separated by $< 2\Theta$, (L, U)

In all cases the length of the search area is equivalent (2Θ) for each dimension, unless more tightly bounded by the original problem formulation.

5.3 Finding a Good α for Large Problems

Of all the parameters of the various CC algorithms, the feasibility distance tolerance α is the one that has the most effect on performance. Choosing a low value of α ensures that a point will be improved as much as possible before passing it to a local solver, but this may be costly in terms of computational effort. Not only may there be a point where additional CC iterations become less efficient at moving towards feasibility than a given nonlinear solver would be, but in a multistart context one would like to move on from unpromising points quickly. Due to these considerations, a larger α may be desired.

Previous experiments [6, 15] with varying α looked mainly at solving to feasibility with CC alone, and let it be fixed when comparing specific solvers. Once one has chosen a specific solver to pair with, however, it is useful to perform a one-time specific study to find the best value for the parameter for that solver.

The design of this study involves taking a small number of representative problems to study in depth. For each problem a number of test runs N is made from a random point, launching a local solver each time a new α threshold is crossed. The test

Inputs:

- An initial feasibility tolerance v
- A progress tolerance $prog_{max}$
- A reduction factor r
- Maximum number of iterations μ

1. Generate a random point, P
2. Evaluate P with a local solver, record result
3. Initialize $\alpha \leftarrow v$
4. Repeat μ times:
   - 4.1 IF P meets tolerance α THEN
     - 4.1.1 Evaluate P with a local solver, record result
     - 4.1.2 $\alpha \leftarrow \alpha \times r$
   - 4.2 IF P's progress ratio is less than $prog_{max}$ THEN
     - 4.2.1 Evaluate P with a local solver, record result
   - 4.3 Update $P \leftarrow P'$ by performing a Constraint Consensus iteration, note progress ratio
5. Evaluate P with a local solver, record result

Figure 5.1: Test Run for Evaluating α Performance with a Solver

structure is described in Figure 5.1, where α is progressively reduced by some factor r, and CC allowed to continue until it reaches this new feasibility threshold, up to a maximum number of iterations μ. A solver is launched before any iterations, each time a new α is met successfully, and after the end of iterations.

For each threshold the quantities of interest include the number of runs that reached it successfully, the percentage of solver launches that succeeded from the threshold when it was reached, and the total time taken to reach a successful result. One would like to find the α that minimizes the average total time to solution,

while also maintaining a high probability of predicting success (to avoid unnecessary launches). Once a good α has been established, integration of CC and a specific solver into a multistart algorithm can proceed.

Chapter 6

Algorithm Design for Multistart Constraint Consensus

Having established that Constraint Consensus is useful in a singlestart context (and having discussed its behaviour on large problems), its applications in multistart algorithms will now be introduced. Several point selection possibilities will be evaluated: uniform random, filtered random, latin hypercube sampling (see Section 3.1.1), and a directed random *voting* method developed during the course of this research. The voting method aims to make use of the information encapsulated in the consensus vectors at previously evaluated points, in order to better select the next region of space in which to sample. Because of its demonstrated strength, the randomized standard starting point heuristic (see Section 3.1.3) is incorporated as a "special" point in the voting method.

The essentials of all the variations of Multistart Constraint Consensus (MCC) are covered in Figure 6.1 (note the definition of 'best' and the updating of α will be discussed in detail in Section 6.2). A point P is generated by some method, and then

CC is applied. If the CC invocation returns a point that meets its tolerances, it is passed to a nonlinear solver for evaluation. If the solver returns a feasible solution, the algorithm terminates successfully. If either CC or the solver returns unsuccessfully the point is either stored or discarded, depending on the variation. If after ν solver launches a solution is not found the algorithm exits unsuccessfully.

6.1 Random MCC Methods

To begin with, various simple random multistart methods were evaluated with MCC. The most basic is to simply select N points, apply CC, and launch the solver with the result (successful or otherwise). This will be referred to as *Basic Random* MCC (BR-MCC). A slightly more advanced version is to only launch the solver from points that meet the CC stopping conditions. In this case the best N points are stored as the algorithm proceeds, so α may be relaxed as appropriate (see Section 6.2). This will be referred to as *Filtered Random* MCC (FR-MCC).

6.2 The Voting Method

Instead of sampling randomly, ideally one would like to weight sampling towards more promising zones. As the conceptual basis for the success of CC implies that a consensus vector will usually point towards a feasible region, given a collection of consensus vectors one should be able to make an educated guess about the location of one or more feasible regions. Figure 6.2 depicts the results of five CC invocations on a 2D Branin test function, with the feasible regions enclosed by three roughly elliptical shapes. For each invocation one iteration has been completed (the first in each pair

Inputs:

- a limit, ν, on number of local solver launches, L
- a limit, τ, on CPU time used for MCC work, MCC_w
- a limit, ρ on the number of points N_p to generate before relaxing α
- CC parameters, $\alpha, \mu, prog_{max}$
- a point generator

1. $L \leftarrow 0$
2. Evaluate a point:
   - 2.1 Generate a Point P (see Figure 6.4) , $N_p \leftarrow N_p + 1$
   - 2.2 Update $P \leftarrow P'$ by applying Constraint Consensus with $(\alpha, \mu, prog_{max})$
   - 2.3 IF P meets CC tolerances THEN:
     - 2.3.1 Apply a local solver to P
     - 2.3.2 $L \leftarrow L + 1$, $N_p \leftarrow 0$
     - 2.3.3 IF P is a feasible solution THEN
       - Exit successfully.
   - 2.4 Store or Discard P (see Figure 6.7)
   - 2.5 IF $N_p \mod \rho = 0$ THEN
     - 2.5.1 $\alpha \leftarrow$ | longest feasibility vector of best unlaunched stored point |
     - 2.5.2 $P \leftarrow$ best unlaunched stored point
     - 2.5.3 Apply a local solver to P
     - 2.5.4 $L \leftarrow L + 1$
     - 2.5.5 IF P is a feasible solution THEN
       - Exit successfully.
3. IF $L < \nu$ AND $MCC_w < \tau$ THEN
   - Go to Step 2
4. Exit unsuccessfully

Figure 6.1: Multistart Constraint Consensus - High Level Algorithm

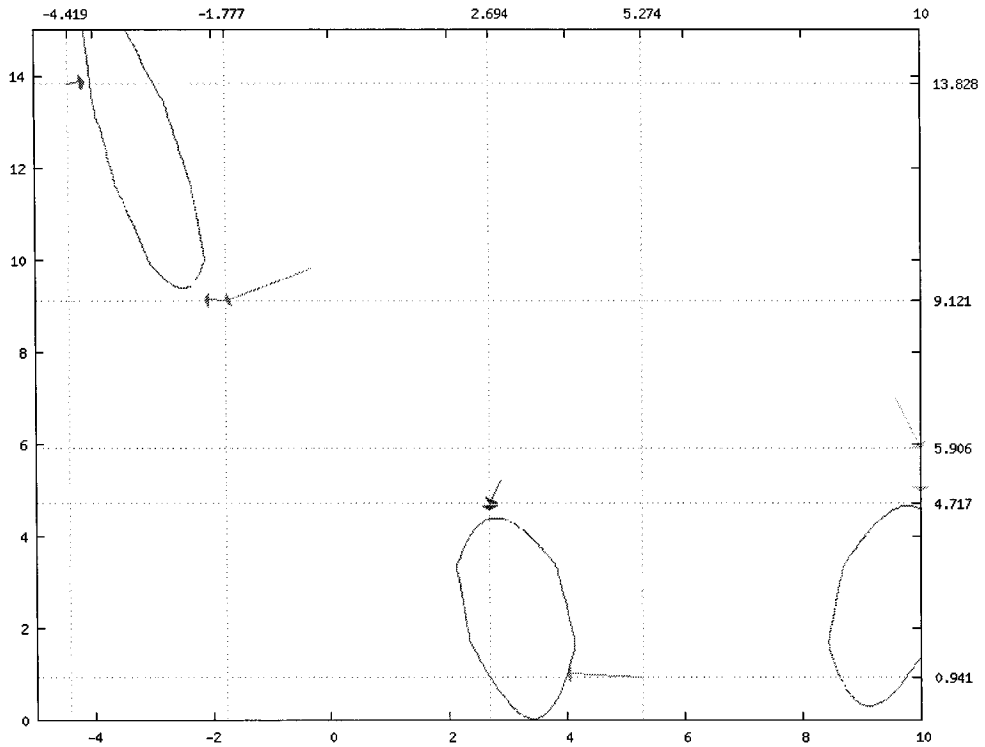

Figure 6.2: CC Invocations on a 2D test function

of arrows), and then a second consensus vector calculated (the second arrow). It can be seen that in each case the second consensus vector points roughly in the direction of one of the feasible regions.

Of course, it may be desirable to make more than two CC iterations before giving up on a given invocation. For the sake of clarity, we will refer to the point from which the last consensus vector is calculated as a *marker point.* Axis-parallel lines (both vertical and horizontal) through these marker points divide the space into boxes. To identify the next location at which to try a sample point, the natural impulse is to find the box towards which the most consensus vectors point. Unfortunately, as the number of dimensions rises, the number of hyperboxes will rise exponentially.

To avoid this combinatorial explosion, the *Voting MCC* (V-MCC) method works dimension by dimension.

This contrasts with the MSNLP method in a few important ways. MSNLP is interested in *excluding* areas that have already been searched, while MCC is interested in *including* promising areas for feasibility. Also, the use of hyperspheres in MSNLP assumes that all dimensions can be treated equally, which may not be valid for many problems. Dimensions may have wildly different scales (e.g. μC and kV), or some may just have narrower areas that contain feasible solutions than others. This is another advantage of working dimension by dimension. Lastly, hyperspheres are unable to cover space completely without overlapping, with the issue becoming more pronounced at higher dimensions. The rectangular areas used by V-MCC do not have this problem.

To identify the best ranges in which to sample for a given dimension, the axis is divided into *bins* at the appropriate coordinate value of each of the marker points. Into these bins *votes* will be placed, as determined by the final consensus vectors at each marker point. A *pointer* consists of a coordinate value paired with the corresponding component of the consensus vector for one dimension of a point (e.g. a point P's x_1 value paired with the x_1 component of the consensus vector at point P).

Figure 6.3 shows the updating of vote totals for the bins in one axis upon the addition of a new pointer (in gray). Each pointer places votes in all the bins it points towards, until it reaches a pointer that points in the opposite direction. Following this rule, the gray pointer adds one vote to the bin to its immediate right, as well as the next two bins, before finding a pointer in the reverse direction. The longer arrows above the diagram indicate how far each pointer votes, and so the number of arrows above a bin is equal to the number of votes in that bin.

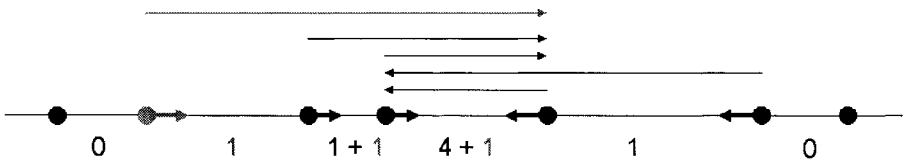

Figure 6.3: Vote Placement on a Single Dimension

In order to use these votes to find a new point x, a random selection is made from the bins for each dimension in turn, weighted by the vote totals. Vote totals may either be left as is, or be scaled prior to the weighted random selection. Squaring was found to be a reasonable choice for emphasizing regions with high totals, without overly skewing the selection. A coordinate value for x_m is then chosen randomly within the selected bin in each axis, with the results combined and returned as a newly generated point in step 2.1 of Figure 6.1. As storing every point generated would lead to exponentially increasing computational costs, only N points are used to perform this calculation at each MCC iteration. Because the number of points is limited, the store or discard step (2.4) attempts to replace the current "worst" point with the new point only if it is "better." The metric used will be described in the details section below. To avoid stagnation of the point set, if after λ attempts to replace the worst point a better point has not been found, it is replaced regardless (by the best point found in the λ attempts).

A description of generating a point via the voting method is provided in Figure 6.5. However, some method of generating the initial N points is needed. Figure 6.4 describes the state machine-like behaviour of point generation for V-MCC. First, the standard heuristic point is returned. Secondly, a latin hypercube sample of $N - 1$ points is generated and returned one by one. If none of these points has reached

feasibility, the next point can be generated by the voting method, as there are now N points with which to construct the markers, bins, and pointers. The maintenance of the set of N points via the store or discard step is summarized in Figure 6.7.

A further detail (to be justified by experiments in Section 7.5.1), is that of 'relaxing' the feasibility tolerance α after every ρ iterations. Depending on model characteristics, CC may not be able to reach the desired α even after a large number of invocations. For a consensus vector to meet the feasibility tolerance, the length of all feasibility vectors must be shorter than α. So the length of the longest feasibility vector at a point is the smallest α for which it will be considered to meet the tolerance. For this reason α is relaxed to the longest feasibility vector of the best (unlaunched) point found so far every ρ iterations, to allow launches to proceed (starting with that best point).

6.2.1 Voting Details

With the overall structure of the algorithm explained, the handling of the bounds and reduced starting area should be described in greater detail. The first issue is how bins on either extreme of the axis (e.g. the two bins with 0 votes in Figure 6.3) are dealt with. If these *boundary bins* contain votes, they indicate a trend to sample outside of the hyperbox containing the current points. The interpretation used in initial experiments considered a boundary bin to represent everything from the dimension bound (possibly positive or negative infinity) to the closest marker point on that axis. If that bin was selected for a new coordinate by the voting algorithm, a random value was duly generated in this (potentially quite vast) region. This was identified as a potential problem. For a model with 1000 dimensions and a 1% chance per dimension of generating one of these large coordinate values, a point with 10 extremely large

Inputs:

- Initial Mode ← heuristic
- An empty set of points RandomPoints
- A number of points N

1. IF Mode = heuristic THEN
   - 1.1 Mode ← lhs
   - 1.2 RandomPoints ← a latin hypercube sample of size $N - 1$
   - 1.3 Return the randomized standard heuristic point
2. IF Mode = lhs THEN
   - 2.1 Remove a point from RandomPoints
   - 2.2 IF RandomPoints is empty THEN Mode ← voting
   - 2.3 Return the removed point
3. IF Mode = voting THEN
   - 3.1 Return a point using the voting method in Figure 6.5

Figure 6.4: Generating a New Point For V-MCC

Inputs:

- bin vote totals, $votes_{i,m}$, for $i = 0 \ldots N$ bins, and $m = 1 \ldots M$ dimensions
- coordinates $coords_{i,m}$, for $i = 1 \ldots N$ marker points, and $m = 1 \ldots M$ dimensions
- a boundary bin width *binwidth*

1. For each dimension m in M:
   - 1.1 $t = \sum_{i=0}^{N} (votes_{i,m})^2$
   - 1.2 $s =$ A random integer on $(1, t)$
   - 1.3 $v = 0$
   - 1.4 $bin = 0$
   - 1.5 REPEAT:
     - 1.5.1 $v \leftarrow v + (votes_{bin,m})^2$
     - 1.5.2 IF $v > s$ go to Step 1.6
     - 1.5.3 $bin \leftarrow bin + 1$
   - 1.6 IF $bin = 0$ THEN
     $low \leftarrow coords_{0,m} - binwidth$
     ELSE
     $low \leftarrow coords_{bin-1,m}$
   - 1.7 IF $bin = (N + 1)$
     $high \leftarrow coords_{N,m} + binwidth$
     ELSE
     $high \leftarrow coords_{bin,m}$
   - 1.8 $x_m \leftarrow$ A random real number on $(low, high)$
2. Return generated point x

Figure 6.5: Generating a New Voted Point

values will likely be produced at every iteration. Based on the experimental results of CC's performance over large distances of higher dimensional space (see Section 7.3), these points are unlikely to be successful on the large majority of problems, and overall progress will be hindered.

To allow for a greater chance of success, the initial points are chosen in a reduced starting area, as described in Section 5.2. However, as models do exist whose solution is outside this area, CC invocations are allowed to move outside this area, as are points generated by the voting method. To allow the voting generated points this freedom, while still keeping them within a reasonable area, the bins on either end of each axis are given a fixed size.

This size (*binwidth* in Figure 6.5) is set to the average distance between the initial points (i.e. the average initial bin width). This is clarified in Figure 6.6, which depicts a starting area of magnitude twenty units (the dashed box) for one dimension, divided every five units by a dotted line. As they would be on average, the initial 4 points are placed in the centre of their hypercube segments, making each bin 5 units wide, as expected. The size of the boundary bins is fixed to this *binwidth* (unless this would exceed the bounds, in which case they are reduced accordingly). In this example, the variable is assumed to be doubly unbounded, and the outer edges of the boundary bins (marked by the angle brackets) are at -12.5 and 12.5. If votes placed in the boundary bins lead to new points being generated here (and they meet the storage criteria), the outer marker points will move outward in space, moving the boundary bins with them. This allows the potential for success in problems whose feasible region(s) do lie outside of the space defined by the reduced starting area.

The vote recalculation steps 1.3 and 4.3 in Figure 6.7 follow largely the procedure described above to tally the votes. However, several special cases must be noted.

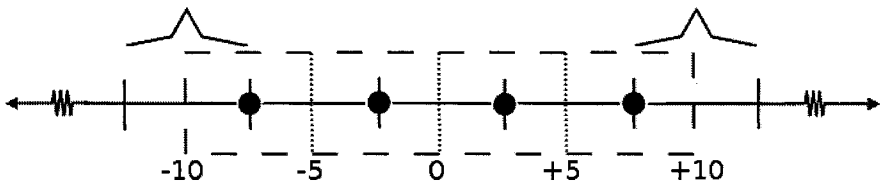

Figure 6.6: Example of Initial Boundary Bin Placement

For instance, if a pointer Q_n (i.e. consensus vector component/coordinate pair) at coordinate value x points in *neither* direction, it places a vote in the bins on either side of itself. The immediate area around x meets the feasibility criterion for some values of the other variables, so sampling nearby is to be encouraged. Pointers voting towards x will place their last vote in the bin on their side of x, as they have a higher confidence that the bin on the near side is feasible than that on the far side.

In addition, it is presumed to be desirable to bias selection away from points where either a solver has been launched and resulted in failure or CC has finished with numerical errors. Such points will place *negative votes* in the bin on either side of them (currently one in each). The use of negative votes brings up more questions to be dealt with in future work.

The possibility of a negative vote total in a bin also requires that all votes be adjusted, in order for the weighted random selection to work correctly. The lowest negative value among all bins in a dimension is found, and then all the bins are adjusted up by that same value. The minimum acceptable value can be made 0 (to exclude some regions) or 1 (in order to always allow a small possibility of sampling all areas). The current implementation uses 0 as the minimum, after some initial experimenting with both values.

Finally, as often occurs in NLP algorithms, some measure of relative 'goodness'

of a point needs to be established, in this case to perform the comparison in Steps 1 and 2 of Figure 6.7. Points are compared based on two factors:

1. a point where fewer constraint evaluations resulted in error is better than a point with more errors, and

2. if two points have the same number of errors, the point whose longest feasibility vector is the shortest is better.

Note that as mentioned above the second item can also be thought of as the smallest value of α that this point satisfies (as the magnitude of all feasibility vectors at this point will be equal or less than this value). As α is the primary criterion for evaluating whether a point should be passed to the solver, this is a satisfying measure of point quality.

6.2.2 Termination Conditions

A maximum number of solver launches ν is the primary stopping condition for these experiments, in order to complete the tests in a reasonable amount of time. Further, the total amount of time spent on CC iterations and voting calculations was limited to τ. The number of points generated N_p was also limited for some of the larger tests.

6.2.3 Initial Points

Before any points can be generated based on vote totals, clearly some initial points are necessary to supply the votes. Given its high rate of success, the randomized standard heuristic point should always be the first point tried. Then, in order to get a relatively even view of space, a latin hypercube of $N-1$ points is made, for a total

Inputs:

- a collection of N *points* with a worst point P_{worst},
- a new point P_{new},
- the current candidate P_{cand} to replace P_{worst},
- the number of attempts n_{worst} to replace the current P_{worst},
- the maximum number of attempts λ to replace P_{worst}

1. IF P_{new} is better than P_{worst}
   - 1.1 Remove P_{worst} from *points*
   - 1.2 Add P_{new} to *points*
   - 1.3 Recalculate the vote totals
   - 1.4 $n_{worst} \leftarrow 0$
   - 1.5 Exit
2. IF P_{new} is better than P_{cand}
   - 2.1 $P_{cand} \leftarrow P_{new}$
3. $n_{worst} \leftarrow n_{worst} + 1$
4. IF $n_{worst} > \lambda$
   - 4.1 Remove P_{worst} from *points*
   - 4.2 Add P_{cand} to *points*
   - 4.3 Recalculate the vote totals
   - 4.4 $n_{worst} \leftarrow 0$
   - 4.5 Exit

Figure 6.7: Storing or Discarding a Point with the V-MCC Method

of N initial points. These points are passed to the CC implementation (and then the local solver, if appropriate) one at time, and if any reaches feasibility the algorithm terminates. Otherwise the voting algorithm is initialized.

Based on the success reported in [21] on artificially bounding variables with large bounds, and confirmed by the study described in Section 5.2, the initial points of the latin hypercube are chosen in a small area around the standard heuristic point. Individual CC invocations are free to move within the original bounds (i.e. outside the initial starting area), and voting iterations may move outside the starting area as well. In this way the algorithm first explores the most probable area for a feasible point, without restricting later phases of the search. This 'inside-out' approach contrasts with methods of starting with large bounds and slowly reducing them.

Chapter 7

Experiments: Setup and Parameter Finding

Before testing of the MCC methods as a whole could begin, first the proper settings of several important parameters had to be determined. The setup and results of the experiments described in Chapter 5 will each be discussed.

7.1 Models

Finding a good set of test models is critical to proper development and testing of any feasibility seeking method. Two different sources were used for the models in these experiments, as described below.

7.1.1 COPS

The COPS test set [8] was created as a series of "difficult nonlinearly constrained optimization problems from applications in optimal design, fluid dynamics, parameter

estimation, and optimal control, among others." It consists of 22 different problem types (only 19 of which actually contain nonlinear constraints), most of which have three different problem sizes. For instance, the *Largest Small Polygon* problems have their size varied by setting the number of sides n_v to 50, 100, or 200. The exception is the *Tetrahedral Mesh Smoothing* problems, for which 5 meshes of different shapes are used. This results in a test set of $18 \times 3 + 5 = 59$ models with nonlinear constraints.

Tests presented below that required a small but representative test set of large models were run on the 'small' versions of the various problem types, with the first mesh smoothing model (tetra1) arbitrarily representing that problem class.

7.1.2 COCONUT

The second primary source of models is the COCONUT (COntinuous CONstraints, Updating the Technology) Benchmark, part of a project funded by the European Union in order to encourage development of new algorithms across several related disciplines [27]. Their test set includes 1322 models in three formats (including AMPL: A Modeling Language for Mathematical Programming), and incorporates the GAMS GLOBAL library [11], the LIA constraint satisfaction test problems [33], and the CUTE test collection (which itself includes the Argonne test set, the Hock and Schittkowski collection, the Dembo network problems and the Gould quadratic programs, among others) [13]. The problems come from a wide variety of fields and incorporate many different function types, and as such are very useful as a heterogeneous test set.

7.2 Software and Hardware Environment

Development of research software necessarily requires a balance of flexibility and verboseness with efficiency and speed. For instance, the various CC methods are descendants of a common abstract class, so that any method can be selected via command line parameters at runtime. This is necessary to allow comparisons between the various methods, while sacrificing some efficiency by not hard coding and optimizing one path through the code.

In the interest of collaboration and future reuse of code, the coding was carried out in the Microsoft Visual Studio 2003 environment [25]. The MCC code was written in C#, and made use of Professor Chinneck's AMPL Reader interface in Visual Basic [5]. These two pieces of code co-exist in the .NET Framework's virtual machine (version 2.0 of which was used [24]), and thus can communicate efficiently. Unfortunately, the low-level AMPL hooks [1] for connecting to solvers are written using C features that cannot yet be easily ported into the .NET environment, so a performance penalty is potentially being paid with every constraint evaluation call. Regardless, CC iterations as implemented are still measured as remarkably cheap time-wise, and would only be faster in a more optimized implementation. Any performance penalty will only reflect negatively on the MCC methods versus other methods.

Individual solver launches are performed by copying the model file to a temporary directory, adding the desired starting coordinates and parameters to the model, and then passing a script to launch AMPL on this model. KNITRO is launched in a separate process from AMPL, and it is the time used by this process that is measured via the Windows API to discount any overhead from AMPL. Also, the AMPL presolver is disabled to prevent any confusion about the number of variables and constraints,

as well as changes to the bounds.

The bulk of the experiments were run on a Windows XP PC with a 3.2 GHz Intel Hyperthreaded processor with 1 GB of installed RAM. The software used was single threaded, so the times reported are based on using one of the two 'virtual' CPUs on the processor. CPU time is measured rather than clock time, to reduce the effect of any background processes (which would mostly utilize the second virtual CPU, regardless).

7.3 Setting The Progress Tolerance Parameter

In order to find a good value for the progress tolerance $prog_{max}$, the changes in consensus vector lengths were recorded over successive CC iterations. First, two problems were explored in detail with 20 random initial points each (see Figures 7.1–7.4). It was found that each launch behaved quite similarly, and that as expected DBavg was the best performing CC variant. Also, the shape of the curves for the length of the consensus vector (a variable space measure) and the sum of the infeasibilities (SINF, a constraint space measure) were very similar for DBavg in particular, and can be used essentially interchangeably for evaluating the algorithm's progress.

Next, 30 CC iterations starting from a random point within the bounds were recorded for each of the models in the small COPS test set. The magnitude of their consensus vectors is plotted in Appendix A.1. Of interest is finding the iteration at which the performance is starting to plateau. Several metrics for identifying the 'knee' in the curve were considered, but the progress tolerance was found to be both simple and effective. The point at which the magnitude of each consensus vector exceeds 85% of the previous vector's magnitude (i.e. a progress ratio of 0.85) is marked on

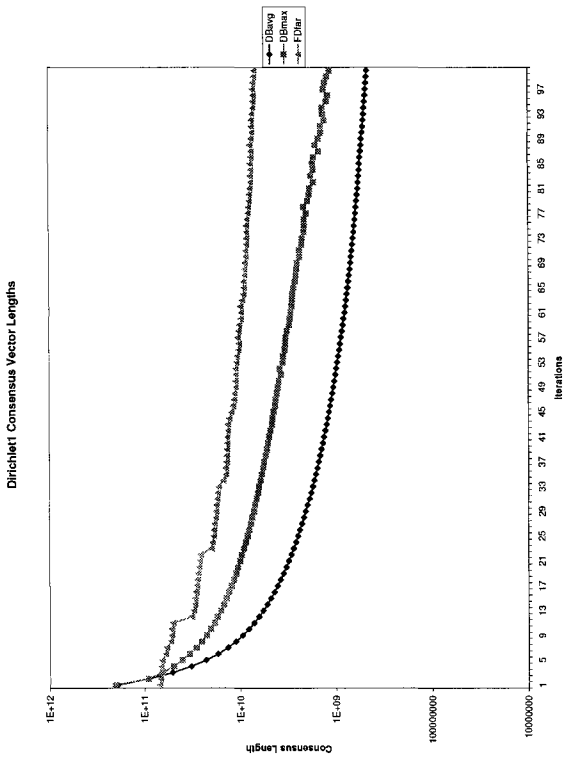

Figure 7.2: Averaged Consensus Vector Lengths Per Iteration for Dirich1

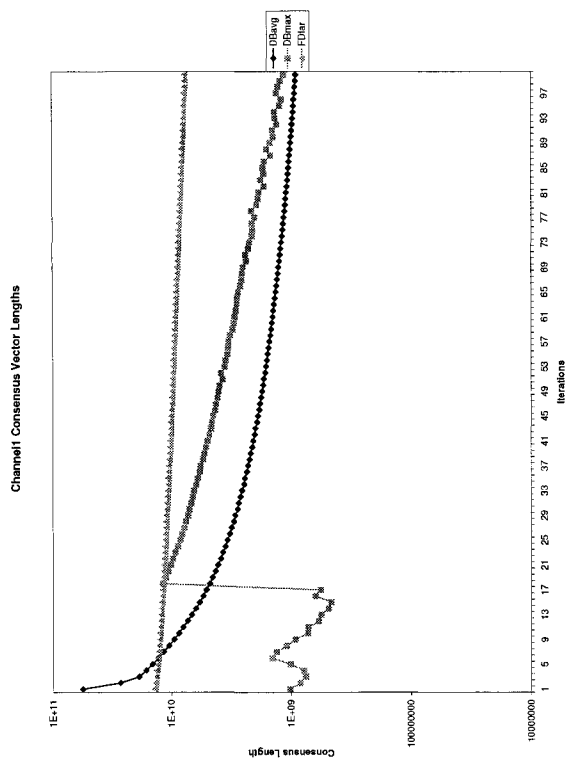

Figure 7.1: Averaged Consensus Vector Lengths Per Iteration for Channel1

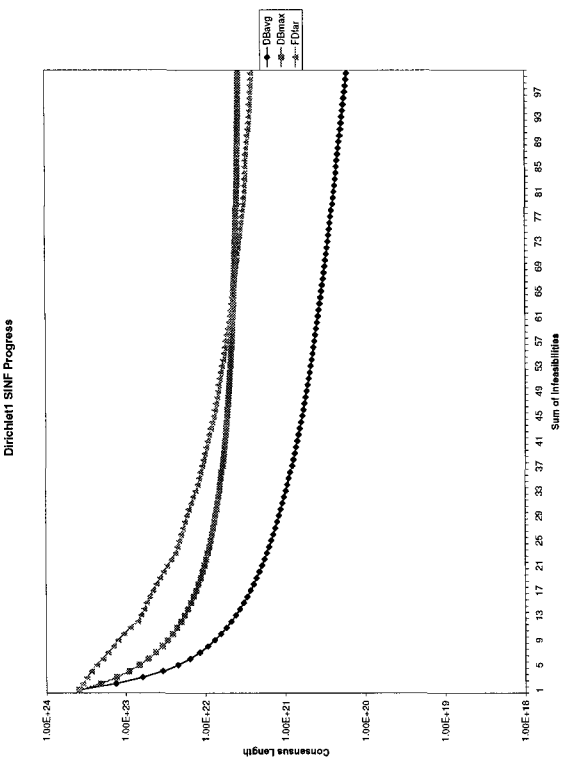

Figure 7.4: Averaged SINF Per Iteration for Dirich1

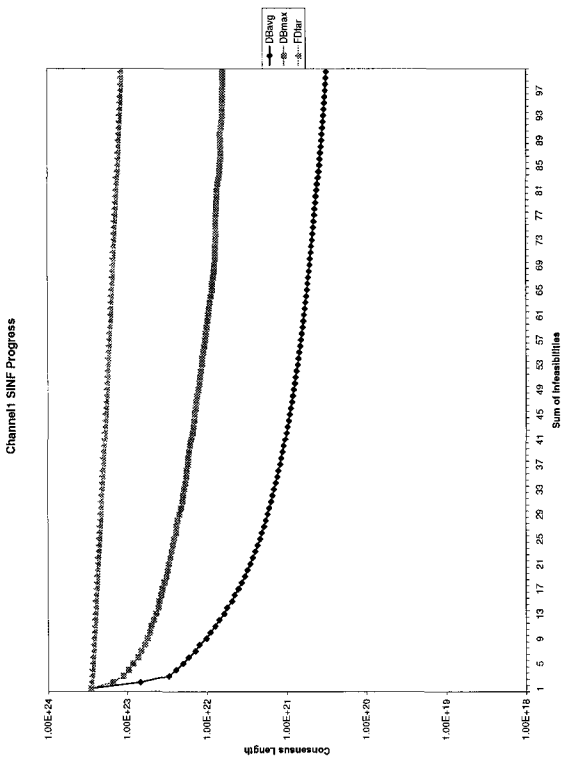

Figure 7.3: Averaged SINF Per Iteration for Channel1

each chart with a larger diamond point, and it can be seen that it does a good job of identifying the beginning of each run's plateau.

An interesting note is that three of the models (polygon1, robot1, and tetra1) do not follow a strictly decreasing curve like the others. These three have a higher than average percentage of both nonlinear constraints and inequality constraints, and so this behaviour may be due to an increased likelihood of constraints flipping between active and inactive on successive iterations. Extending the progress ratio to be a moving average over several iterations may help to smooth out this behaviour, but the increase in complexity was not deemed necessary given the good performance of the simple metric.

Also of note is that given the performance plateau, CC generally seems to be able to move only about two or three orders of magnitude closer to feasibility (the exception being elec1, which is able to solve to the feasibility tolerance without triggering the progress tolerance. Its other exceptional attribute is that *all* of its constraints are nonlinear). As such it is highly desirable that initial points can be chosen with high probability of being within a few orders of magnitude of a feasible region. In the following sections an area where a feasible point is highly probable to be found will be established.

7.4 Identifying a Reasonable Start Area

As described in Section 5.2, there is some evidence to show that limiting starting points to an area smaller than that of the bounds can increase the chances of solver success, both in optimization and feasibility seeking. In order to empirically determine a reasonable starting box, the minimum distance to feasibility problem proposed in

Equation (5.1) was solved for a large set of problems. Specifically, all models in the COCONUT set that could be loaded by the student edition of AMPL (less than 300 variables and less than 300 constraints), for a total of 992 models. These problems were solved with the student version of KNITRO 5.0.2 on a P4 2.53 GHz system with 1 GB of RAM, running Microsoft Windows 2000. KNITRO was given 30 minutes of CPU time from 5 different points in the problem space in order to find a solution. 18 of the 992 models (1.8%) failed to find a feasible solution within these limits, so there is a small margin of error on the results. Distances are measured against the standard heuristic point, as it has been shown that a random area around this point is the most promising at which to start. Note that since KNITRO returns a local optimum, it is possible that there are closer feasible points for some models.

The magnitudes of the distances from the standard heuristic point to the closest feasible point are summarized in Figure 7.5. Several points are of note:

- For 96.5% of the models the average distance to the closest feasible point was less than 1×10^4 per dimension
- For 95.5%, a maximum distance of $< 1 \times 10^4$ was recorded over all the dimensions
- For distances over 1×10^4, the improvement in the percentage of models who's maximum is covered in the range is 1% or less per order of magnitude
- A distance of 1×10^3 per dimension contains the maximum of 91.7% of models and the average of 94.1%, the first significant drop as the area around the heuristic is reduced

Given this information, an area $\pm 1 \times 10^4$ around the heuristic point (or $+2 \times 10^4$ for lower bounded variables and -2×10^4 for upper bounded variables) seems like a

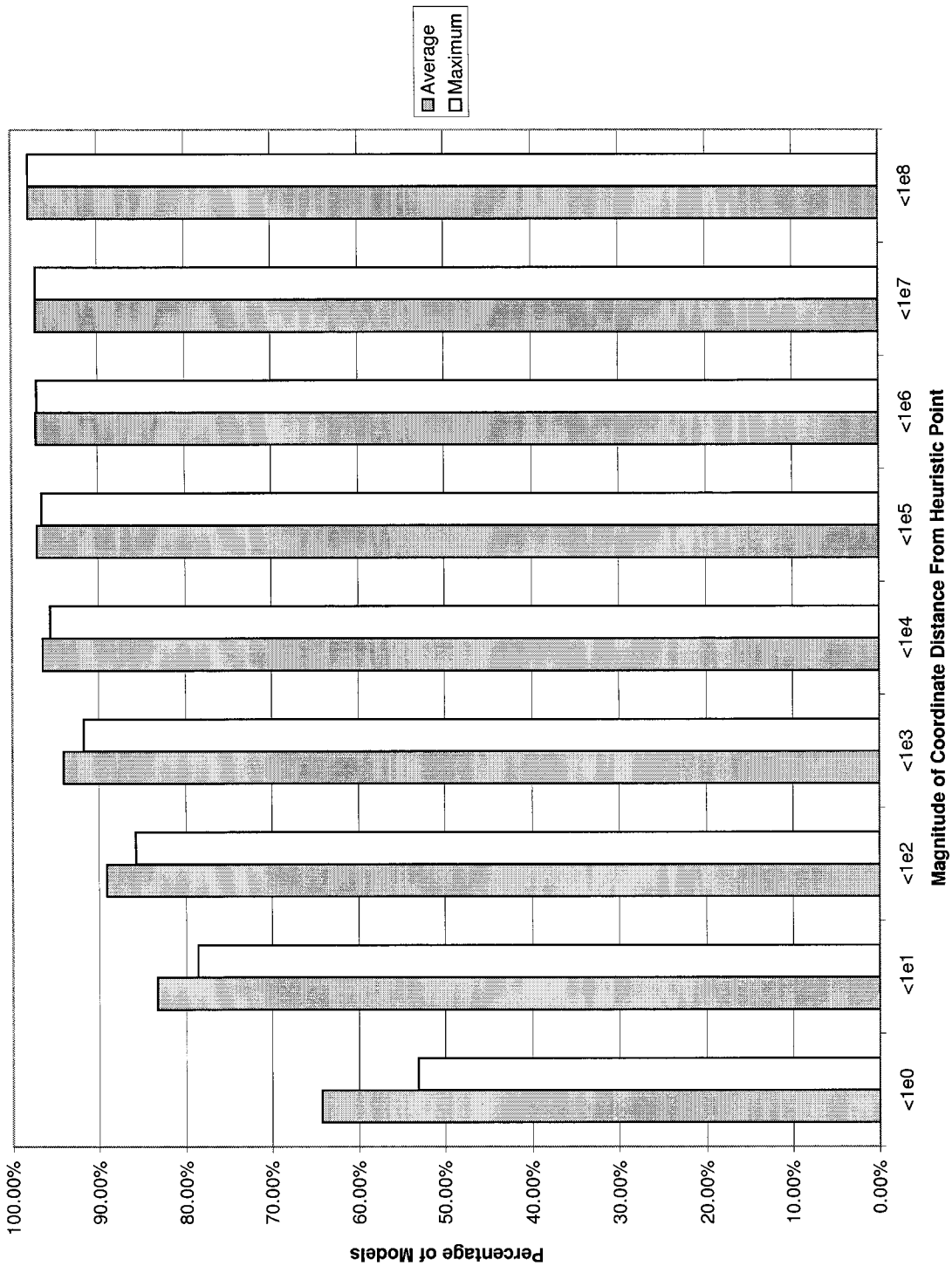

Figure 7.5: Coordinate Distance of Closest Feasible Point to Standard Heuristic Point

reasonable area in which to place initial points on the vast majority of problems, as the improvement per order of magnitude plateaus past this point. Furthermore, CC can be expected to cover a few orders of magnitude distance on those problems with more distant feasible regions before even passing on to a local solver, as demonstrated in the previous section.

7.5 Local Solver—KNITRO

Although CC can be used to solve problems fully to feasibility, it is generally recommended for use as a pre-solver. Selection of the nonlinear solver to do the final solve on a point is thus quite important. In the results published in [15], the KNITRO solver [37] emerged as the the most successful in terms of finding feasibility from a given point. Given its strength at reaching feasibility it was selected in order to solve the maximum number of problems, although it reduces the possible room for improvement for the MCC pre-solver.

The *Full Edition* of KNITRO 4.0.2 was used with default settings, except where noted. The output level was set to 1, presenting only summary statistics at the end of a run. In most cases the time limit was set to 10 or 20 minutes in order to keep the experiments reasonable, and the default iteration limit was increased from 10 000 to 1 000 000 to keep time as the major limiting factor. Some of the smaller problems still occasionally reached the iteration limit, but due to the apparent lack of progress it was considered more efficient to let the solver start again with a different point. All other settings were kept at defaults. It is worth nothing that KNITRO makes use of three different algorithms, and attempts to select the best for a particular problem automatically. No attempt was made to influence this behaviour, in keeping with the

motivation of a naive modeller.

A note must be made on the interpretation of KNITRO's return status codes, as they are presented in the documentation and interpreted by the AMPL interface. The major categories are:

solved - the final point meets KNITRO's feasibility and optimality tolerances (default 1×10^{-6}).

solved? - the point is not quite feasible, but progress per iteration has slowed to a crawl (default movement tolerance is 1×10^{-15}). KNITRO thinks the point may nevertheless be optimal, in the sense that it may be the closest point to feasibility in the (local) space.

infeasible - the algorithm halted at an infeasible point.

failure - the point could not be evaluated (numerical error).

limit - the problem reached the iteration or time limit. AMPL returns the same code for each, so cases were programmatically separated by looking at the returned processing time (and was later verified using the saved output).

The key point is the distinction between the definitely solved and the 'near-solved' (return code of "solved?") problems. Solution existence is not known *a priori* for all problems, so in some cases the near-solved results may in fact be as close to feasibility as is possible. As this cannot be established easily (and in some cases the modeller may be able to accept the reduced tolerance), this category is treated separately as a possible solution.

7.5.1 Finding a Good α for KNITRO

As described in section 5.3, it is important from an efficiency standpoint to find the right value of α for a given solver. Smaller values almost universally increase the likelihood of solver success, but potentially at the expense of computational efficiency. In addition to various α values, the progress metric discussed in Section 7.3 will be evaluated as a possible indicator of when CC should hand off a point to a local solver.

Given the assumption that the V-MCC method will eventually generate points within a reasonable range of the feasible region, it was decided to investigate the performance of points near the feasible region on models with a known feasible region. Also, as a very large number of solver launches per model were necessary, only a small number of models could be evaluated. To this end, those problems of the "small" COPS set that solved from the heuristic point where chosen, for a total of 14 problems to be thoroughly examined.

For each model, 10 points were generated by perturbing each coordinate of the heuristic point by a random value on $(-1 \times 10^4, 1 \times 10^4)$. For each of these points, the procedure described in Figure 5.1 was followed, with an initial feasibility tolerance $v = 8192$, $prog_{max} = 0.85$, reduction factor $r = \frac{1}{2}$ and maximum number of iterations $\mu = 100$. This allowed the performance of the CC/KNITRO combination to be evaluated with α values of descending powers of two (until 100 iterations were reached), as well as at the initial and ending points, and at the point indicated by the progress tolerance stopping condition. The length of the longest feasibility vector was recorded at the point marked by the progress tolerance and also at the final point, as an indication of what α value would have allowed them to be launched. The initial point was considered to have $\alpha = \infty$. To save time, if a point met a newly reduced α with no additional iterations KNITRO was not relaunched, as the results would be the same.

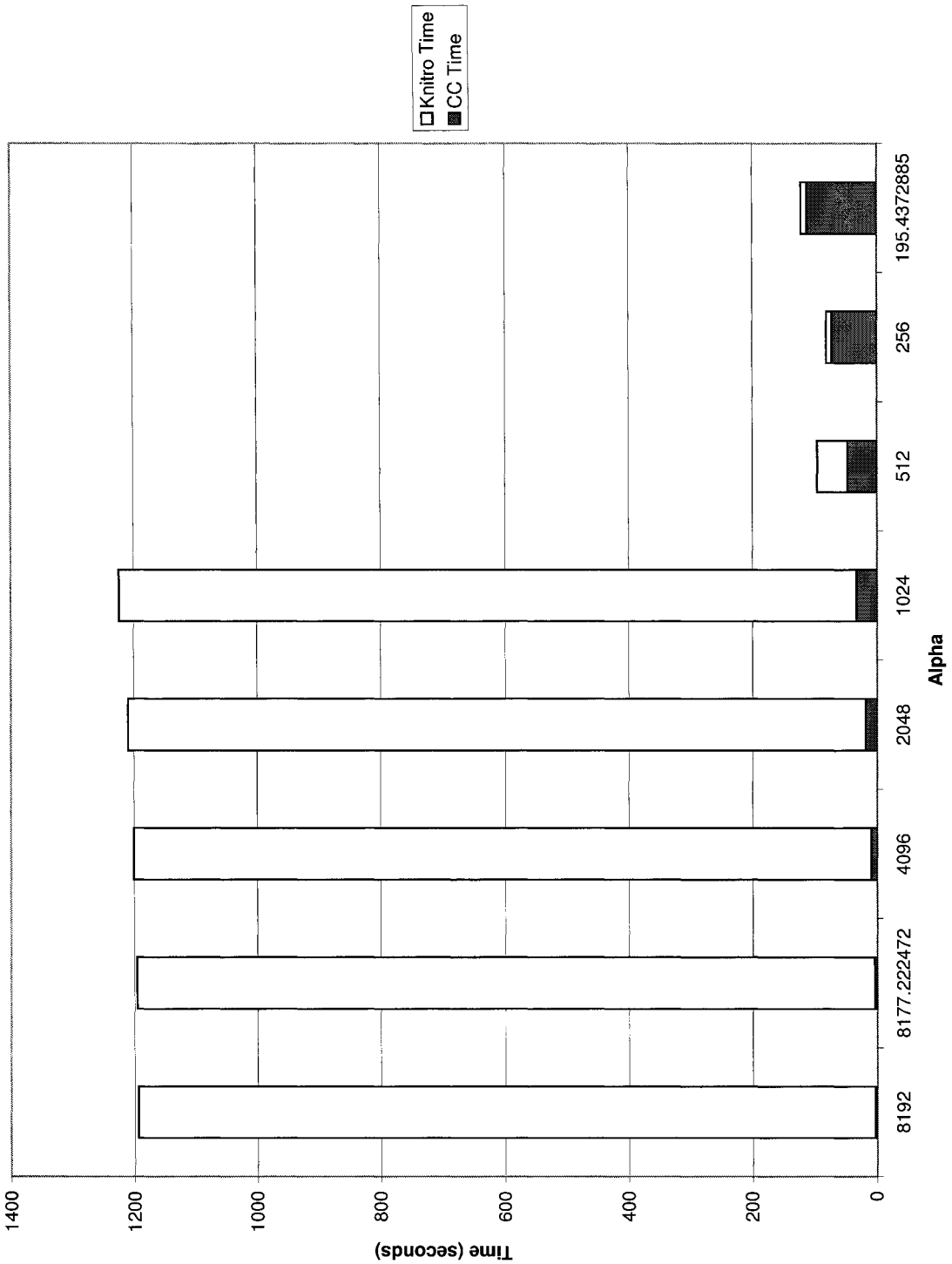

Figure 7.6: Example of Time Taken by Launching KNITRO at Various α Values, Run #4 on Catmix1

Figure 7.6 shows a good example of the performance trade-off investigated here. With higher values of α (8192-1024), KNITRO is not able to reach feasibility at all with its full 20 minutes of allotted time. With $\alpha = 512$, after 46.6 seconds of CC computation, KNITRO is able to reach feasibility in 49.6 seconds (a total of 96.2 seconds). When α is reduced to 256, feasibility is reached after 71.9 seconds of CC time and 9.6 seconds of KNITRO time (a total of 81.5 seconds). However, $\alpha = 128$ could not be reached, and even though the final point has a maximum feasibility vector length of 195.4, KNITRO is still only able to solve the problem in 9.6 seconds, despite 112.3 seconds of CC computation.

To approximate the 'best' value for each model, the value of α that resulted in the shortest total CC and KNITRO solution time (i.e. 256 in Figure 7.6) for each run was recorded, then the median value taken. The medians for the 14 problems are summarized in Figure 7.7. Of note:

- The progress tolerance $prog_{max}$ did not prove to be an especially good predictor of solver success. Unlike α, it does not itself measure distance to feasibility. If the initial point is far from feasibility, it may be triggered far from the feasible region.

- The three problems with the extremely high values (16384-18835) all have nearly identical problem structure, with 1900+ linear equalities and only 21 nonlinear inequalities (linear constraints, especially equalities, can lead to poor performance for CC). These problems were only able to reach $\alpha = 4096$, and KNITRO performed much the same from all values. As such the recorded median values are from the point identified by the progress tolerance. These results indicate that a higher α may be desirable for highly linear problems.

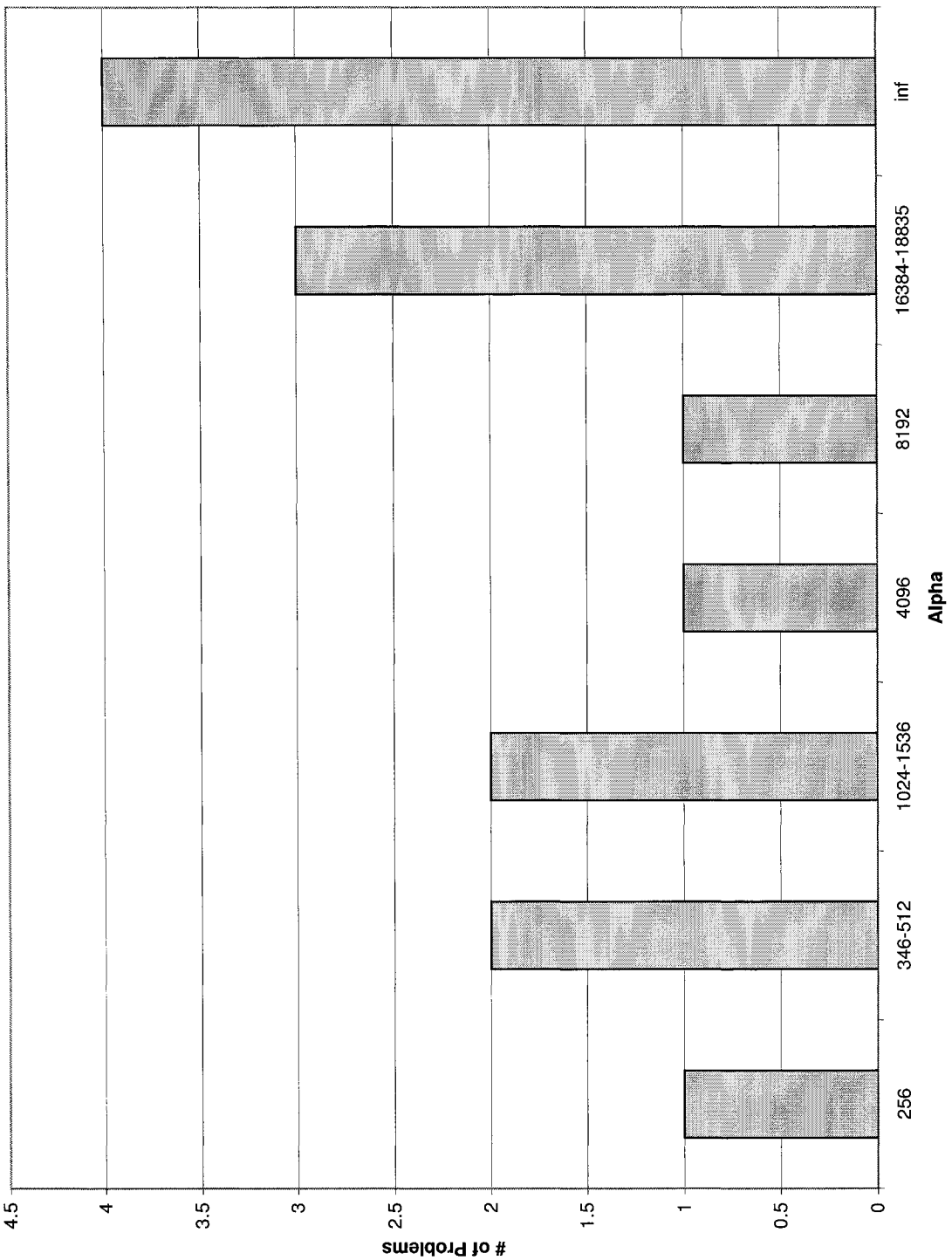

Figure 7.7: Best α Value by Lowest Median Solution Time

- Three of the four problems whose median value was ∞ were within α tolerances down to around 1 with no CC iterations for all points. A negligible amount of computational effort (< 0.01 seconds) to verify this caused the initial launch to be considered faster. This implies that any reasonable α value is essentially equally fast for these problems. However, one of these problems (marine1) did take a significant amount of effort per CC iteration, with the effect of α being somewhat random.

- Other than the above notes, the best α value, performance wise, clusters in the mid hundreds to low thousands.

The second major performance metric is how well a point meeting a certain α tolerance predicts solver success. The ranges for various success rates are depicted in Figure 7.8, with the darker regions having the highest percentage of KNITRO success given an α in that range is met, and the lighter the lowest. For instance, the polygon model had a 100% success rate for points satisfying α values ranging from just under one to the mid-hundreds, 40-59% for those in the high hundreds, and 0-19% for values up to the tens of thousands. Blank ranges are those which could not be reached by any of the CC launches for a given model. Based on this information, values in the low hundreds seem to be the best predictor of success, with $\alpha = 256$ (the dashed line in the figure) showing a slight advantage in falling in high percentage ranges over $\alpha = 512$ (the solid line).

This is more clearly shown in Figure 7.9, which averages over all launches for all models. The two lines represent two different treatments for models who were not able to reach a given α value for *any* launches—the lower line treats them as a 0% success rate, whereas the top line averages only over the models for which that

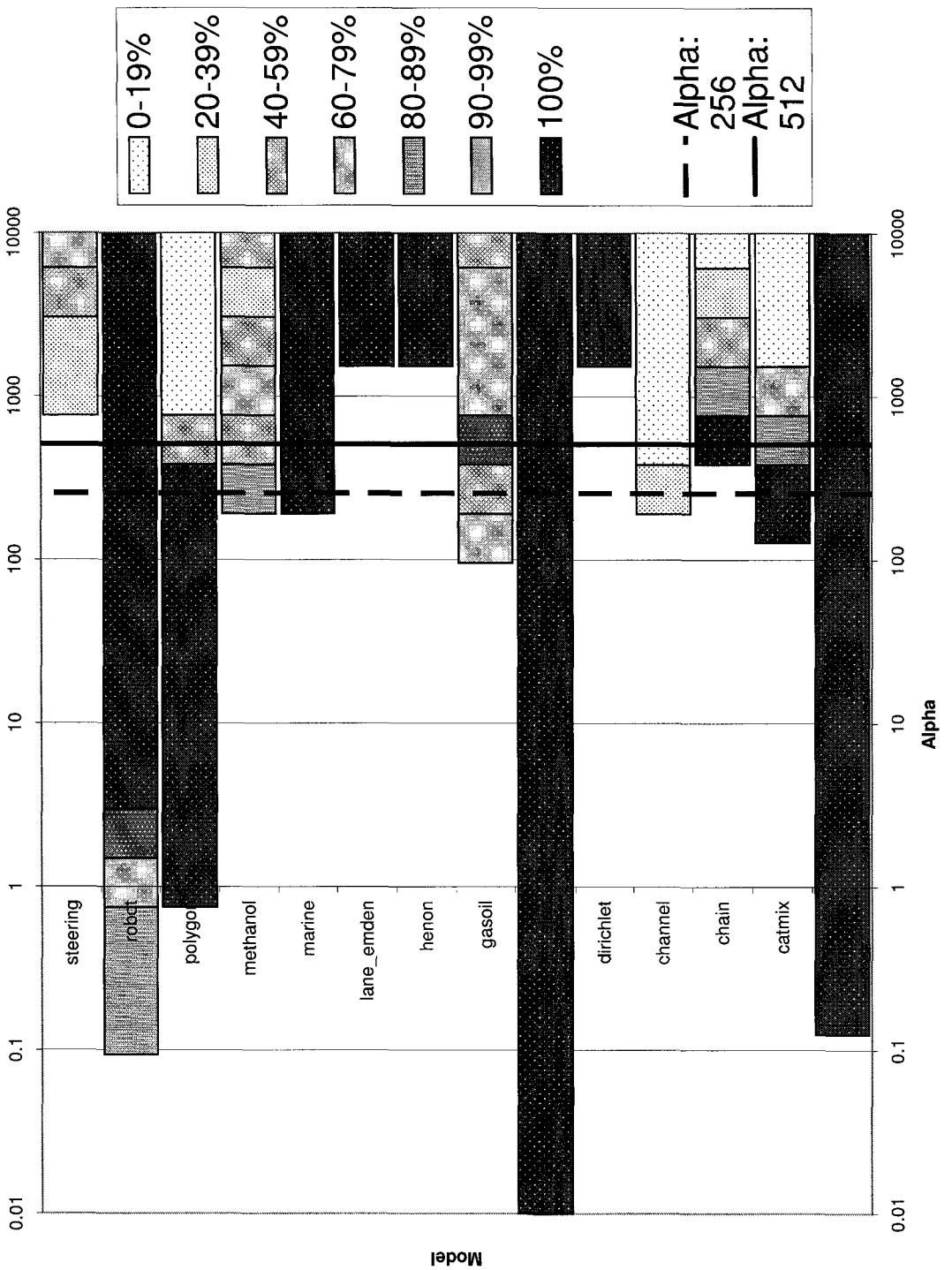

Figure 7.8: Best α Value by Rate of Solver Success

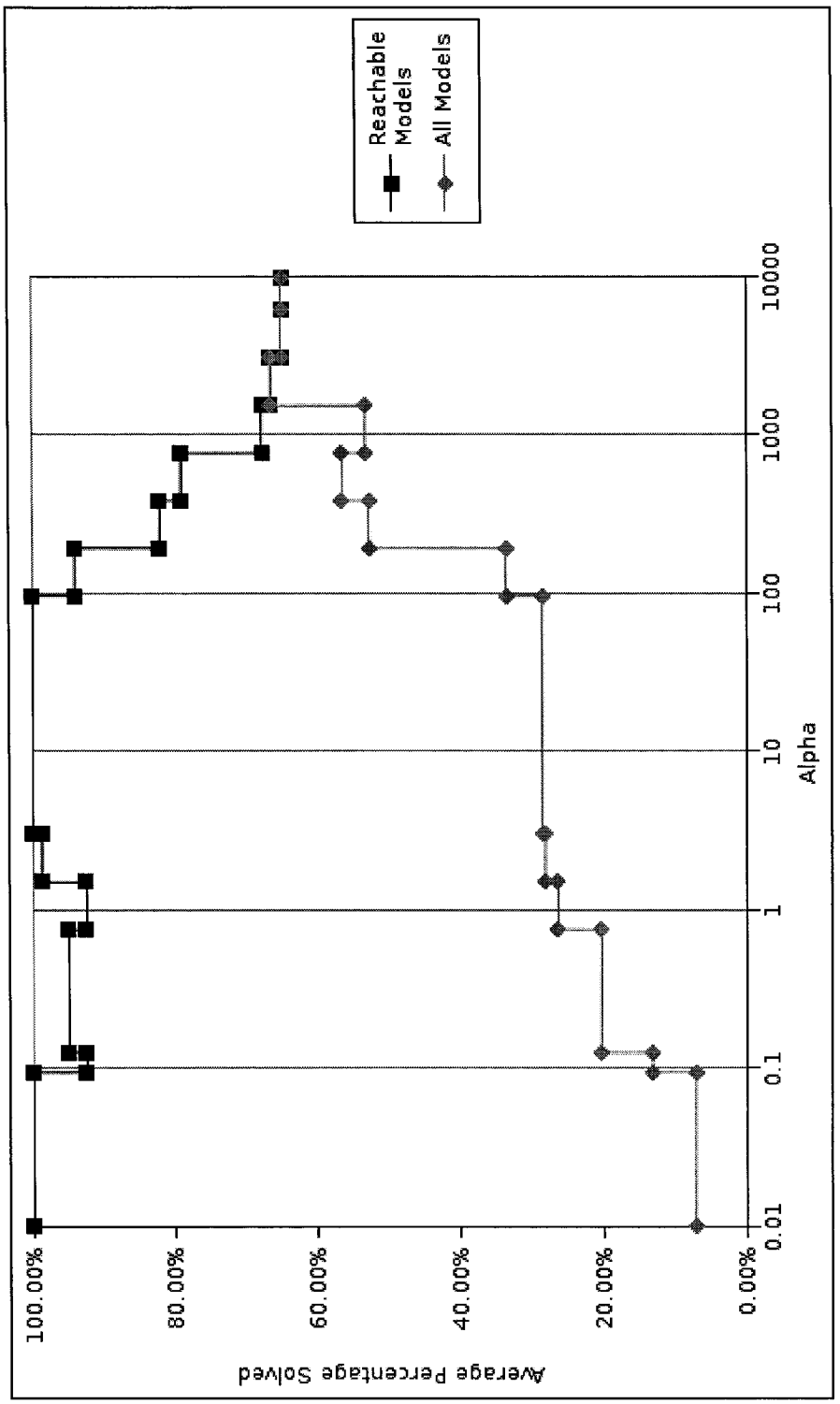

Figure 7.9: Averaged Solver Success Rates by α

feasibility tolerance was reachable. The two meet in the thousands, where all the models were successful in reaching that α. Values in the low hundreds are the lowest for which more than half of the models were reached, and show a jump in the overall average solved. Reaching even lower values demonstrates the greatest confidence in predicting solver success from a given point, but far fewer models can reach these values. 256 was selected as the intial value of α, as it is reachable for a good number of models (9 out of 14), and still has a success prediction rate of 82%. However, some problems can be solved from even larger α values (see especially lane_emden, henon, and dirichlet) that cannot reach lower values. This observation led to the addition of the ρ parameter in the FR-MCC and V-MCC methods, wherein α is set to 256 to begin with (due to its high predictive rate), but successively relaxed if it cannot be reached in a reasonable number of CC invocations. In this way the method avoids 'wasting' solver launches on problems where low feasibility tolerances are attainable by CC, while still being able to solve those where they are not.

7.6 Setting The Voting Parameters

Having established the best CC parameters to use, the correct parameters for the voting portion of V-MCC needed to be established. Specifically, the value of N is the dominant parameter to evaluate. As the combination of the randomized heuristic point and CC find feasible solutions for a great number of models, and this is incorporated as the first stage in V-MCC, it is useful to only evaluate those models which fail to reach a feasible point from the heuristic point. Of the entire COPS set, 11 models meet this criteria, and were used for initial tuning.

Firstly, a range of N values were tried with a fixed α (256), and a maximum of

100 points were evaluated with CC. Initial starting points were chosen in the area determined in Section 7.4, and $prog_{max}$ was set to 5, allowing α to dominate the filtering of points to pass to the solver as recommended by Section 7.5.1. Any of the heuristic point, the initial latin hypercube points, or the voting generated points were launched if they met the α tolerance.

Table 7.1 lists the number of problems solved for each N value, as well as a comparison of the average number of launches and time spent for the problems that were solved for *all* values of N. As this is only two problems in this case, further validation of the results is necessary. Although an N value of 15 took a lower average time and number of launches, the values above 10 were not able to solve quite as many problems to feasibility. An N value of 5 has a very low average time among those with the higher success rate.

In order to increase the chances of solver success further, a series of tests was then run wherein α was progressively relaxed, as recommended above. After every 20 points generated without solver success, α was changed to the length of the longest feasibility vector of the best stored point (i.e. the lowest value of α that would allow the best point point not yet tried to meet the feasibility tolerance). This best stored point was launched, and voting continued as before for another 20 points. As this test would produce additional launches and take longer to run, only N values below 10 were used. As can be seen in Table 7.2, the higher success rate for lower values was maintained, and an N value of 5 again offered the speediest performance of the values with the highest success rate. Again this is a comparison of only four problems, so the success of the method will be further vetted in the following chapter.

Table 7.1: Voting Performance at Various Values of N

N	Solved	Launches	CC Time	KNITRO Time	Total Time
3	*3*	6.5	*130.96*	8873.59	9004.55
5	*3*	3	155.59	2746.43	2902.02
10	*3*	5	463.14	4926.8	5389.94
15	2	*2*	334.12	*1527.58*	*1861.7*
20	2	3.5	383.98	3246.21	3630.19

Table 7.2: Voting Performance at Various Values of N, With α Relaxation

N	Solved	Launches	CC Time	KNITRO Time	Total Time
3	*5*	5	*915.23*	5192.84	6108.07
5	*5*	*3*	1936.17	*2491.7*	*4427.88*
10	4	3.75	1438.48	3654.59	5093.07

7.7 Summary of Findings

The first major finding in this chapter was that although CC could make a significant impact on closeness to feasibility, on almost all models its effectiveness tapers after a few orders of magnitude. The progress ratio was used to successfully identify the knee in feasibility performance, which intuitively seems like a good place to stop and launch a local solver. However, given the results of launching KNITRO at various α values as well as the $prog_{max}$ cutoff, it did not appear that $prog_{max}$ was as good an indicator of probability of solver success as α. As the same tapering in CC performance is exhibited independently of the distance of the starting point from the feasible area, it follows that progress is not a good indicator of closeness to feasibility. Instead, α was selected as the preferred metric for deciding whether to launch from a given CC invocation. The important point to note is the estimated feasibility distance of points can only be efficiently improved by a few orders of magnitude, and as such a

reasonably small starting area should be established if there is to be hope of success. It was decided to keep the $prog_{max}$ check in the algorithm with a higher value (5), to catch wildly diverging invocations.

Next, parameters for the V-MCC method were evaluated. Given the variability in α values reachable by CC for various models, it was proposed and tested that successively relaxing α improved the performance of V-MCC. Also, it was found that a relatively small number of points (5) was needed for efficient and effective vote calculation. As the setting of these voting parameters was done on a very small set of 11 models, it is important to verify the effectiveness of V-MCC with these parameters in the following chapter.

Chapter 8

Results

Having established good CC parameters for large nonlinear models, a reasonable initial search area, and preliminary MCC parameter tunings, MCC's performance can be verified against another test set.

8.1 COCONUT Test Set—Revisited

As discussed in section 7.1.2, the COCONUT test set [27] is a large collection of heterogeneous linear and nonlinear models. Running on the entire set of problems is clearly impractical, so steps were first taken to identify the most difficult problems for use as a test set. In keeping with the objective of following the course of a naive modeller, all problems with at least one nonlinear constraint were first tried by launching KNITRO from the origin (with coordinates adjusted to the closest variable bound if 0 is out of bounds) and the standard heuristic (nonrandom) initial point (see section 3.1.3). Problems that KNITRO solved successfully from either of these points within 10 minutes were rejected. Next, 20 random points were provided to KNITRO

with the same time limit, and the 122 models solved to feasibility from one of these points were set aside for efficiency testing.

The 50 problems that remained were divided into problems that failed to find either feasible or near feasible solutions (which we will refer to as *infeasible* in the charts below), and those that returned the *solved?* result on at least one invocation. These 50 problems are used for effectiveness testing. They were further categorized by the number of nonlinear constraints M_{NL}, those with 100 or less, those with 101-1000, and those with more than 1000. Evaluation of nonlinear constraints and their gradients is the most expensive operation in CC, and therefore a good predictor of problem difficulty. This gives a total of six problem categories—three sizes with 'infeasible' and 'solved?' subdivisions.

Also note that the problems used for testing herein are completely disjoint from the problems earlier used to tune the method parameters. As with most methods, no set of parameters will give the best possible performance on every model, but the best parameters for a model cannot be known *a priori*. It is therefore important to demonstrate that the recommended parameter settings established with the tuning problems work well on a different set of problems, without first retuning for the new set.

8.1.1 Parameter Settings

Solver settings were kept uniform across all the methods, with the time per KNITRO launch capped at 10 minutes and all other settings set as described previously. Each method was allowed a maximum number of launches ν of 20. In the case of the V-MCC method, the number of launches permitted for points generated by the voting process was set to 20, regardless of the number of launches performed from the heuristic or

latin hypercube points. The total number of launches for the successful V-MCC runs (or the initial identification of a near feasibile solution) was always within the first 20 overall launches, so the limit could equally have been set to 20 for the method as a whole with no effect on the results.

The CC parameters were set to $\mu = 100$, $\alpha = 256$, $prog_{max} = 5$ (to allow α to dominate as the criteria for launching the solver), and α was relaxed every 20 generated points in the case of FR-MCC and V-MCC (e.g. $\rho = 20$). The total time τ allowed for CC, voting and filtering work was two hours, the total number of points N_p generated by FR-MCC and V-MCC for CC invocations set to 100, the maximum number of times to attempt to replace the worst stored point with a better point was set to $\lambda = 10$. A reduced starting area of magnitude 2×10^4 or less per dimension was used, as determined in Section 7.4.

8.1.2 Effectiveness Evaluation

Due to the conglomeration of methods that make up the V-MCC method, the results must be compared in several different ways. The first and most important point to establish is that V-MCC as a whole is more effective than the simpler FR-MCC and BR-MCC methods (and that all are better than the purely random multistart used during the problem selection). Table 8.1 presents the percentages of problems that succeeded or produced possible best solutions (*solved?* return codes), broken down by problem size (number of nonlinear constraints, M_{NL}) and their reason for selection during the initial screen (failure to find even near feasible solutions, or only near feasible solutions). The actual number of models represented by the percentages is given in brackets. For the problems that returned only infeasible results during the screen, V-MCC dominates at all problem sizes, finding more definitively feasible

solutions for more models than FR-MCC and BR-MCC, even if possible solutions are considered acceptable. BR-MCC does slightly better at finding definitive solutions for previously only possibly solved problems in the 101-1000 range (given that there are only 5 problems in this category, chance likely plays an element), but otherwise V-MCC dominates in finding solutions and possible solutions for the possibly solved problems as well. It is also worth nothing that V-MCC is the only method that returns a feasible or possibly feasible solution for *all* the problems the screen found to have possibly feasible solutions, so there is no drop in effectiveness. Also, as expected, FR-MCC performs slightly better than BR-MCC, although not in all cases.

Table 8.1: Relative Effectiveness of MCC Methods

M_{NL}	Screen Result	MCC Result	BR-MCC	FR-MCC	V-MCC
1-100	infeasible	solved	33% (4)	50% (6)	75% (9)
	(12 models)	solved?	0.0% (0)	8.3% (1)	0.0% (0)
	solved?	solved	0.0% (0)	10% (1)	70% (7)
	(10 models)	solved?	80% (8)	80% (8)	30% (3)
101-1000	infeasible	solved	33% (2)	33% (2)	50% (3)
	(6 models)	solved?	17% (1)	17% (1)	33% (2)
	solved?	solved	60% (3)	20% (1)	20% (1)
	(5 models)	solved?	40% (2)	80% (4)	80% (4)
1001+	infeasible	solved	13% (2)	20% (3)	33% (5)
	(15 models)	solved?	0.0% (0)	0.0% (0)	0.0% (0)
	solved?	solved	0.0% (0)	0.0% (0)	0.0% (0)
	(2 models)	solved?	100% (2)	50% (1)	100% (2)
Totals	infeasible	solved	24% (8)	33% (11)	52% (17)
	(33 models)	solved?	3.0% (1)	6.1% (2)	6.1% (2)
	solved?	solved	18% (3)	12% (2)	47% (8)
	(17 models)	solved?	71% (12)	76% (13)	53% (9)
	all models	solved	22% (11)	26% (13)	50% (25)
	(50 models)	solved?	26% (13)	30% (15)	22% (11)

Given that V-MCC is more effective than the other MCC methods, now the con-

tribution of the individual elements of the algorithm must be analyzed. Solutions may be found from points generated in 3 ways—the randomized standard heuristic, the reduced bounds latin hypercube, or the iterative voting procedure. If a possibly feasible result is returned, the search is continued in case a definitively feasible solution can be found, so it should be noted that the results in Table 8.2 assign possibly feasible results to the first generation method that returned a possibly feasible solution (even though often more than one was discovered). Percentages in this table are measured out of the total number of problems in the set (50).

Table 8.2: Relative Effectiveness of V-MCC Generation Methods

V-MCC Result	Heuristic	Latin Hypercube	Voting
Solved	46% (23)	2.0% (1)	2.0% (1)
Solved?	8.0% (4)	4.0% (2)	10% (5)

The randomized heuristic clearly dominates in finding definitely feasible solutions, with both the latin hypercube and voting methods only generating points leading to feasible solutions for one model each. They make more of a contribution in locating possible best solutions, locating these for seven models, compared to four found by the heuristic method. It should be noted when considering the relative effectiveness that as the heuristic is the first point tried it tends to catch the "low-hanging fruit," but it is clearly the most effective part of the method. Still, the addition of a multistart allows CC to find feasible or near-feasible solutions to more problems, without reducing the effectiveness or efficiency of the method on problems solvable from the heuristic point.

8.1.3 Efficiency Evaluation

The second main performance metric used to evaluate the MCC method is that of performance—both in terms of total time used per model and the number of local solver launches required. The number of launches is an important metric as it is independent of the efficiency of the MCC implementation, and only takes into account the results of the algorithm. Total time is of course the most important metric in commercial solver software, and as will be shown the MCC algorithm already shows merit by this metric as well.

Table 8.3: Number of Models Solved/(Solved?) by Each Method

Method	$1 \leq M_{NL} \leq 100$	$100 < M_{NL} \leq 1000$	$M_{NL} > 1000$
Random	122	14	15
BR-MCC	117 (3)	13	11
FR-MCC	118 (2)	13	9
V-MCC	118 (4)	14	11 (1)

For this evaluation, the 151 models solved from random points in the screening process (see Section 8.1) are used. Table 8.3 summarizes the number of models solved by each of the methods, with the 'Random' method referring to the random portion of the screening process. The main figures represent problems with a 'solved' result, while those in brackets represent a 'solved?' result. For example, the V-MCC method was able to find a feasible solution to 118 of the models with less than 100 nonlinear constraints, and near feasible solutions to the other four. Both BR-MCC and FR-MCC failed to find either a feasible or near feasible solution for two of the models. Overall, V-MCC found feasible or near-feasible solutions to all of the models except three from the largest category, with the other two methods discovering slightly less.

For a fair comparison, only those models for which definitively feasible solutions were found by *all* methods are included in the following average performance numbers—116 with less than 100 nonlinear constraints, 13 with between 101 and 1000, and 9 with more than 1000.

Table 8.4: Average Performance on 116 Models with $1 \leq M_{NL} \leq 100$

	Launches	CC Time	KNITRO Time	Total Time
Random	1.28	—	35.2	35.2
BR-MCC	1.22	0.01	24.55	*24.6*
FR-MCC	1.21	0.09	25.71	25.8
V-MCC	1.36	0.07	31.1	31.2

Table 8.5: Average Performance on 13 Models with $100 < M_{NL} \leq 1000$

	Launches	CC Time	KNITRO Time	Total Time
Random	1.08	—	111.12	111.1
BR-MCC	1	41.39	0	*41.4*
FR-MCC	1	56.86	0	56.9
V-MCC	1.38	16.51	92.28	108.8

Table 8.6: Average Performance on 9 Models with $M_{NL} > 1000$

	Launches	CC Time	KNITRO Time	Total Time
Random	2.22	—	761.04	761.0
BR-MCC	1.89	1430.66	969.49	2400.2
FR-MCC	1.89	1410.59	969.52	2380.1
V-MCC	1	18.36	18.38	*36.7*

For both the small and medium problem sizes, all the MCC methods outperform the random screen without CC in terms of total time. Somewhat surprisingly, V-MCC uses the highest average number of launches in both cases, leading overall time

to suffer despite a smaller amount of time spent on CC evaluations. The fact that these models were known to not solve from the origin or the non-randomized standard heuristic point, but *did* solve from random points, may help to explain the relative efficiency of the more random CC methods. As the randomized standard heuristic point will be relatively close to the non-randomized variant, this may be in some sense a bad place to start for this set of models (and as it is the first point tried, explains a number of the failed launches). However, it is worth re-emphasizing that V-MCC was still more effective in finding solutions and possible solutions.

Table 8.7: Breakdown of Feasible Solutions Contributed by each V-MCC Component

M_{NL}	Heuristic	LHS	Voting
1-100	82	29	7
101-1000	9	1	4
1001+	10	0	1
Total	101	30	12
% of Total	70.6%	21.0%	8.4%

On the set of larger problems, V-MCC shows its merit more clearly. Although FR-MCC and BR-MCC have a slight advantage over the initial random test in terms of number of launches, they use more time for each launch, and also spend a very large amount of time on CC iterations. V-MCC is the clear efficiency winner, thanks to the randomized standard heuristic (see Table 8.7 for a breakdown by algorithmic component of the 139 feasible solutions found by V-MCC), although all struggle to find feasible solutions for as many problems as the initial screening points.

8.2 Summary

As expected, the randomized standard heuristic point is able to find feasible solutions to a significant fraction of difficult problems, when combined with CC and a strong local solver. The V-MCC method as a whole found solutions to 25 of the 50 hardest models screened from the 1322 model COCONUT test set, and near solutions for 11. Simply combining CC with a random approach found feasible solutions to problems that a random approach could not, and filtering the random points based on their ability to meet the α progress tolerance further improved effectiveness.

For problems where a random search was known to be able to produce feasible solutions where the non-randomized standard heuristic point did not, the incorporation of the randomized standard heuristic point had a slightly detrimental effect on the efficiency of V-MCC for small and medium sized models. However, it retained its edge in effectiveness, and all MCC methods outperformed a random search. On larger problems (those with more than 1000 nonlinear constraints), V-MCC regained its performance edge, while maintaining its effectiveness edge (largely due to the randomized standard heuristic). Given its strong effectiveness and relatively efficient performance, V-MCC is the recommended MCC point generation method.

Chapter 9

Conclusions

Based on the results above, several important conclusions can be drawn. The MCC methods developed over the course of the paper incorporates a series of effective initial point generation methods, which when combined with the DBavg Constraint Consensus method and the KNITRO local solver are able to find feasible solutions to many difficult problems. Increased efficiency was also demonstrated on models solved by a simple random multistart and KNITRO. The recommended MCC method, V-MCC, combines the randomized standard heuristic point, latin hypercube sampling, and a voting method to generate initial points. The most important CC parameter is α, which should be set to 256, and progressively relaxed if needed.

As expected from previous research, the combination of the randomized initial starting point heuristic, CC and the KNITRO nonlinear solver solves an enormous number of nonlinear programs of all varieties regardless of size, and is the first method attempted in a V-MCC run. It contributes the most feasible solutions to the method.

If the randomized heuristic method fails, next points are tried within a hyperbox surrounding the standard heuristic point, as defined by:

- $0 \pm 1 \times 10^4$ for unbounded dimensions
- $c \pm 1 \times 10^4$ for doubly bounded dimensions, where c is the centre of the bounds
- $l + 2 \times 10^4$ for dimensions with a single lower bound, where l is the bound
- $u - 2 \times 10^4$ for dimensions with a single upper bound, where u is the bound

If the difference in bounds for any dimension is $< 2 \times 10^4$, those bounds should be used in place of those listed above. Into this region, a latin hypercube of $N - 1$ points is placed, and CC applied to each point. Any successful points are passed to the KNITRO solver.

If a feasible point has still not been located, the voting procedure described in 6.2 should then be iteratively applied, with an N value of five (with α relaxed every twenty iterations to the length of the longest feasibility vector of the best unlaunched point yet found). Again any points which meet the α feasibility tolerance are passed to KNITRO.

This layered combination of methods has been shown to find solutions to 50% of the most difficult problems in the COCONUT test set, compared with 22% for PR-MCC and 26% for FR-MCC. Models that could not be solved from twenty random points in ten minutes of solver time are able to be solved in seconds with judicious application of V-MCC, and it is the most effective method evaluated here.

9.1 Contributions

Several contributions to the art of finding feasible points effectively and efficiently have been made by this research. Understanding of the performance of the fledgling Constraint Consensus method has been furthered, and its usefulness in a multistart

context established. Also, initial point selection for finding feasibility has been further examined. Findings include:

- Several multistart methods incorporating CC have been developed that successfully improve the efficiency and effectiveness of feasibility seeking for large nonlinear programs, the goal set at the beginning of this research. They were shown to have a significant success rate on problems for which the strongest single start method failed, and that a directed random search was more effective than a simple random search with CC applied. MCC was also shown to outperform a pure random start efficiency wise, with the BR-MCC and FR-MCC methods doing slightly better on smaller problems, and V-MCC on the larger. As V-MCC had the highest overall success rate in reaching feasibility, it is the recommended MCC variant.

- It has also been established that for the overwhelming majority of problems (95.5% of the test set), a feasible point lies within $\pm 1 \times 10^4$ of the standard heuristic point. Any random multistart method is advised to start within this area, as adjusted for the actual model's bounds. The *boundary bins* concept has been introduced to allow this area to slowly expand and move across space, while keeping it to a reasonable size. This contrasts with methods that start with a large search space which slowly shrinks.

- Further, three new elements of CC performance were established:

    1. When starting from completely random points in unbounded, high dimensional models, points can only be improved by between two and three orders of magnitude before progress plateaus.

2. The ratio of consensus vector lengths between iterations is a cheap and successful heuristic measure for detecting this plateau in performance.

3. Appropriate settings for the α parameter when paired with a given local solver can be determined that give a good combination of speed and success rate. It was found that values in the low hundreds were appropriate for the KNITRO solver in particular.

The combination of these three results represents important new knowledge in the use of CC to find feasibility for large models.

9.2 Future Research

9.2.1 Constraint Consensus

As CC as a whole is quite a new method, several new avenues of potential exploration were encountered during the course of the research. Refinement of the progress tolerance continues, and it has been suggested that a moving average may provide better results. Also the relative effect of linear vs. nonlinear constraints and equality vs. inequality constraints requires further examination. Problems with a large number of linear constraints (especially equalities), tend to lead to slowly decreasing progress for CC, whereas nonlinear constraints (especially inequalities) tend towards a more erratic but effective progress.

Also, a brief test was made combining aspects of the DBavg and FDfar methods. FDfar as designed makes the moves suggested by the longest feasibility vector in all the variables it involves, and averages all the others as in the original CC method. By instead using DBavg to set the remaining variables, the benefits of both may be

somewhat combined. Testing on one model revealed slightly better constraint space SINF performance per iteration than DBavg, slightly longer consensus vectors than DBavg, and NINF performance similar to FDfar (it tends to satisfy one constraint per iteration, due to its structure). A related idea is to combine the longest N feasibility vectors that don't share any common variables. However, this is somewhat expensive to implement, and some initial probing suggests the longest vectors tend to share at least one variable, negating the effect.

9.2.2 Refinement of Voting Method

Much potential for experimentation remains within the voting framework. It has been suggested that placing negative votes in the bin behind each successful pointer may be advantageous, as a CC iteration has skipped over that space as one without feasibility. The relative strength of the negative votes to influence away from poor areas also merits exploration. For problems with feasible regions outside of the initial search region, it may be worthwhile to dynamically adjust the size of boundary bins to cover space more rapidly. If they get too large, however, it may be difficult for individual CC iterations to succeed in the created bins.

Further experiments into the dynamic adjustment of α (and possibly other) parameters are also recommended. Given the results on the COCONUT test set, it is possible that the parameters were overly biased to large problems due to tuning on the COPS set. In order to preserve the goal of not requiring expert knowledge on the part of the modeller, some quick and reliable method of determining appropriate settings based on problem size (and perhaps ratio of linear to nonlinear constraints) will be required.

For comparability and simplicity, some potential performance enhancements were

not evaluated and can be explored in the future. No form of variable scaling was used to reduce the effect of large variable values dominating the feasibility tolerance calculations. Also, the AMPL presolve option was disabled, which attempts to reduce the bounds and number of variables, to avoid interfering with MCC's bound reduction tests. Lastly, it could be useful for the user to supply a reasonable starting search box themselves, if known. This contrasts with the user supplied bounds, in that they are not a hard limit, but a suggestion.

9.2.3 Termination Conditions

Much of the trade-off in the multistart method involves predicting the best moment at which to give up on a CC or solver invocation. This is primarily a limitation of a single processor system. In a dual processor setup, it may be desirable to launch a solver on CPU 2 with the best point found after every s seconds of CC time on CPU 1. The more processors available, the more solver invocations that could proceed in parallel. Having only one highly efficient, successful run will allow the other CPUs to stop their solver invocations, reducing total time spent. For example, on a 10 CPU system with 10 parallel launches, one launch termination in 10 seconds means only 100 seconds of CPU time were spent, even if 9 of the launches would have taken thousands of seconds. Progress parameter settings will soon have to be reevaluated to take into account the increasing availability of multiprocessor systems.

Full integration with a solver could also potentially allow for more intelligent use of progress metrics to decide when to hand off a point from CC to the local solver. The relatively poor performance of CC on linear constraints, for example, could possibly be mitigated by using other solver routines to deal with these constraints first.

References

[1] Bell Laboratories. Hooking your solver to AMPL. `http://www.ampl.com/hooking.html`, October 2002.

[2] Y. Censor, M. Altschuler, and W. Powlis. On the use of Cimmino's simultaneous projections method for computing a solution of the inverse problem in radiation therapy treatment planning. *Inverse Problems*, 4(3):607–623, August 1988.

[3] Y. Censor, D. Gordon, and R. Gordon. Component averaging: An efficient iterative parallel algorithm for large and sparse unstructured problems. *Parallel Computing*, 27(6):777–808, 2001.

[4] Y. Censor and S. A. Zenios. *Parallel Optimization: Theory, Algorithms, and Applications.* Oxford University Press, New York, USA, January 1998.

[5] J. W. Chinneck. AMPL model reader object. `http://www.sce.carleton.ca/faculty/chinneck/AMPLreader/AMPLObject.html`, January 2004.

[6] J. W. Chinneck. The constraint consensus method for finding approximately feasible points in nonlinear programs. *INFORMS Journal on Computing*, 16(3):255–265, Summer 2004.

[7] G. Cimmino. Calcolo approssimato per soluzioni dei sistemi di equazioni lineari. In *La Ricerca Scientifica*, volume I of *II*, pages 326–333, 1938.

[8] E. D. Dolan, J. J. Moré, and T. S. Munson. Benchmarking optimization software with COPS 3.0. Technical Report ANL/MCS-TM-273, Laboratory for Advanced Numerical Software, Argonne National Laboratory, February 2004.

[9] O. A. Elwakeil and J. S. Arora. Methods for finding feasible points in constrained optimization. *AIAA Journal*, 33(9):1715–1719, September 1995.

[10] O. A. Elwakeil and J. S. Arora. Two algorithms for global optimization of general NLP problems. *International Journal for Numerical Methods in Engineering*, 39(19):3305–3325, October 1996.

[11] Global World. GLOBAL library. `http://www.gamsworld.org/global/globallib.htm`, 2002.

[12] F. Glover, M. Laguna, and R. Martí. Scatter search and path relinking: Advances and applications. In F. W. Glover and G. A. Kochenberger, editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*, chapter 1, pages 1–36. Kluwer Academic Publishers, Boston, 2003.

[13] N. Gould, D. Orban, and P. Toint. CUTEr: a constrained unconstrained testing environment, revisited. `http://cuter.rl.ac.uk/cuter-www/problems.html`, 2001.

[14] R. L. Haupt and S. E. Haupt. *Practical Genetic Algorithms.* Wiley–Interscience, 2nd edition, May 2004.

[15] W. Ibrahim and J. W. Chinneck. Improving solver success in reaching feasibility for sets of nonlinear constraints. *Computers and Operations Research*, 2005. to appear.

[16] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13(4):455–492, December 1998.

[17] W. Karush. Minima of functions of several variables with inequalities as side constraints. Master's thesis, Dept. of Mathematics, Univ. of Chicago, Chicago, Illinois, 1939.

[18] H. Kuhn and A. Tucker. Nonlinear programming. In J. Neyman, editor, *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492, 1950.

[19] M. Laguna and R. Marti. *Scatter Search: Methodology and Implementations in C*, volume 24 of *Operations Research/Computer Science Interfaces Series.* Springer, February 2003.

[20] L. Lasdon and J. Plummer. Multistart algorithms for seeking feasibility. *Computers and Operations Research*, January 2005. submitted.

[21] L. Lasdon, J. Plummer, Z. Ugray, and M. Bussieck. Improved filters and randomized drivers for multi-start global optimization. *Journal of Global Optimization*, May 2004. submitted.

[22] R. Martí. Multi-start methods. In F. W. Glover and G. A. Kochenberger, editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations*

Research & Management Science, chapter 12, pages 355–368. Kluwer Academic Publishers, Boston, 2003.

[23] M. D. McKay, W. J. Conover, and R. J. Beckman. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, May 1979.

[24] Microsoft Corporation. SDKs, redistributables & service packs. `http://msdn.microsoft.com/netframework/downloads/updates/default.aspx`, 2006.

[25] Microsoft Corporation. Visual Studio .NET 2003. `http://msdn.microsoft.com/vstudio/previous/2003/`, 2006.

[26] B. A. Murtagh and M. A. Saunders. MINOS 5.5 user's guide. Technical Report SOL 83-20R, Systems Optimization Laboratory, Stanford University, July 1998.

[27] A. Neumaier. The COCONUT benchmark: A benchmark for global optimization and constraint satisfaction. `http://www.mat.univie.ac.at/~neum/glopt/coconut/Benchmark/Benchmark.html`, 2003.

[28] J. D. Pintér. Continuous global optimization: An introduction to models, solution approaches, tests and applications. *Interactive Transactions of ORMS*, 2(2), 1998.

[29] M. G. C. Resende and C. Ribeiro. Greedy randomized adaptive search procedures. In F. W. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*, chapter 9, pages 219–249. Kluwer Academic Publishers, Boston, 2003.

[30] M. G. C. Resende and C. C. Ribeiro. GRASP with path-relinking: Recent advances and applications. Technical Report TD-5TU726, AT&T Labs Research, December 2003.

[31] R. T. Rockafellar. Lagrange multipliers and optimality. *j-SIAM-REVIEW*, 35(2):183–238, June 1993.

[32] Saittam. Lhssampling.png. `http://commons.wikimedia.org/wiki/Image:LHSsampling.png`, 2005.

[33] D. Sam-Haroud. Constraint satisfaction test problems. `http://icwww.epfl.ch/~sam/Coconut-benchs/`, 2006.

[34] A. E. Sepulveda and L. Epstein. The repulsion algorithm, a new multistart method for global optimization. *Structural and Multidisciplinary Optimization*, 11(3–4):145–152, June 1996.

[35] W. Tu and R. W. Mayne. An approach to multi-start clustering for global optimization with non-linear constraints. *International Journal for Numerical Methods in Engineering*, 53(9):2253–2269, March 2002.

[36] W. Tu and R. W. Mayne. Studies of multi-start clustering for global optimization. *International Journal for Numerical Methods in Engineering*, 53(9):2239–2252, March 2002.

[37] R. A. Waltz and J. Nocedal. *KNITRO User's Manual.* Optimization Technology Center, Northwestern University, Evanston, IL, USA, April 2005. `http://www.ziena.com/knitro.html`.

[38] Y. Xiao, D. Michalski, J. Galvin, and Y. Censor. The least-intensity feasible solution for aperture-based inverse planning in radiation therapy. *Annals of Operations Research*, 119(1–4):183–203, March 2003.

Appendix A

A.1 Consensus Vector Progress Charts

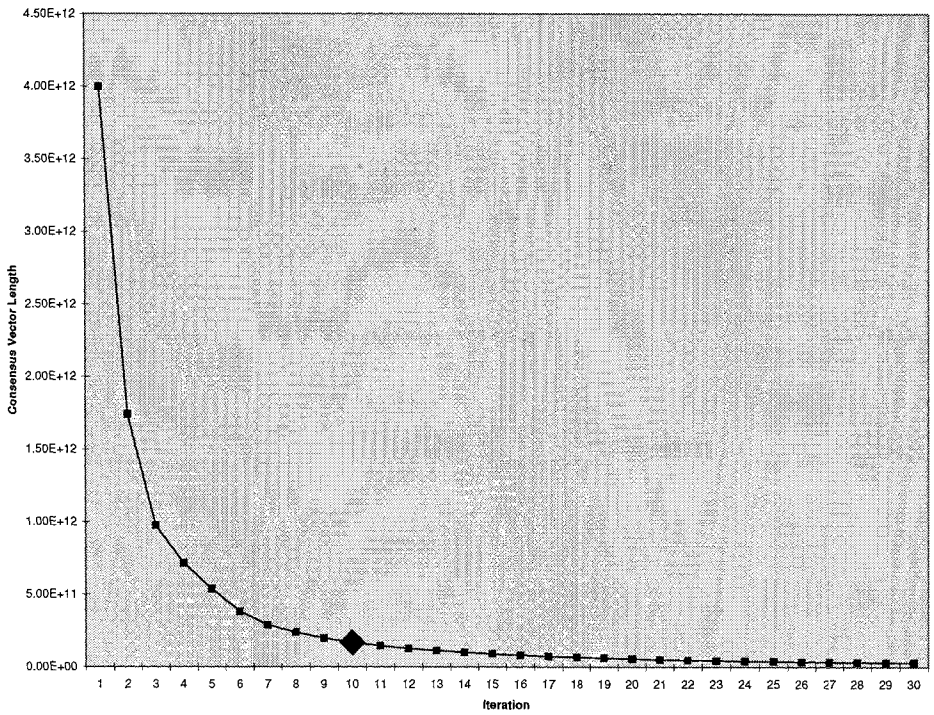

Figure A-1: Progress for Catmix1.mod

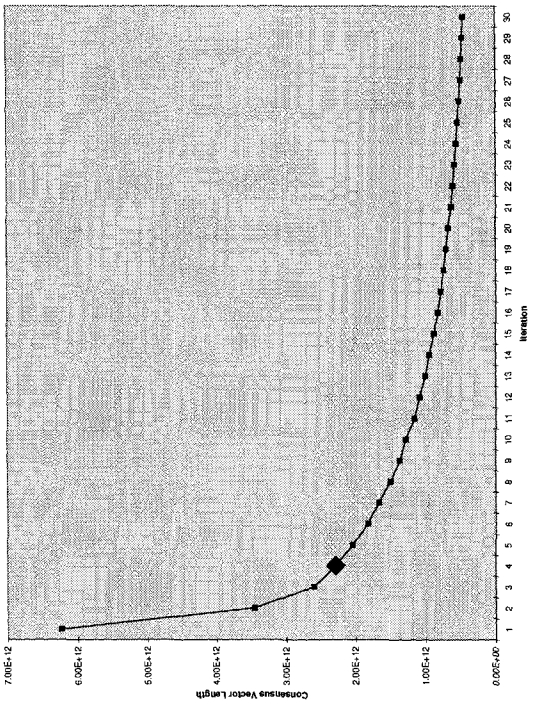

Figure A-3: Progress for Channel1.mod

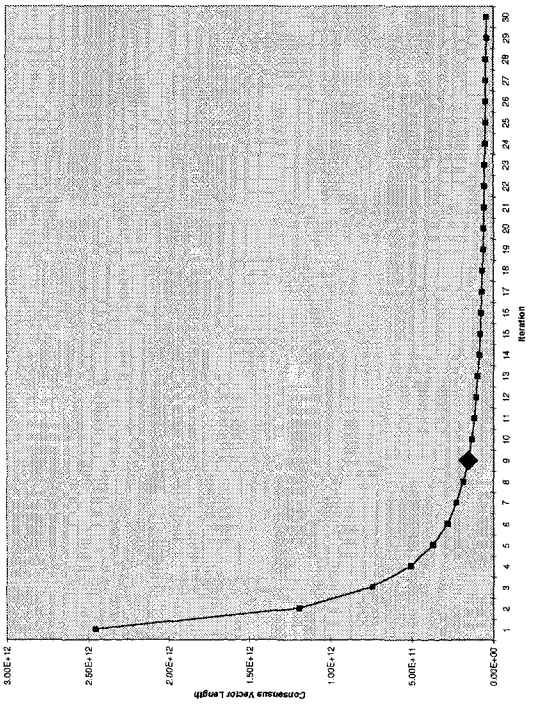

Figure A-2: Progress for Chain1.mod

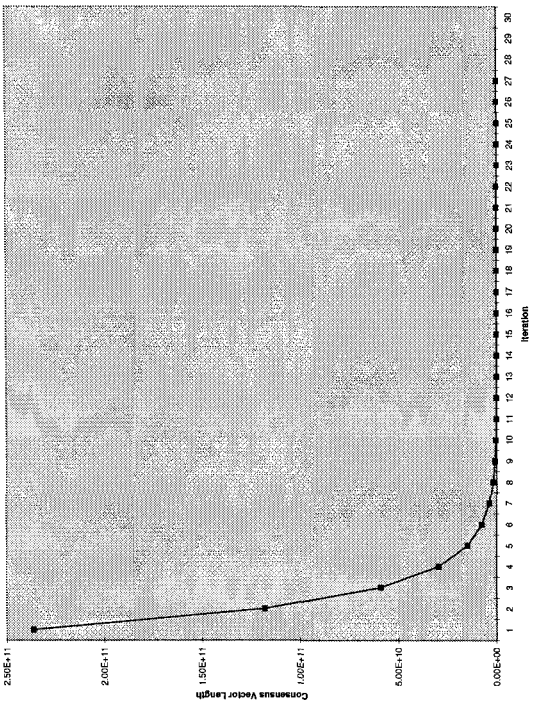

Figure A-5: Progress for Elec1.mod (note this model never triggered the progress tolerance)

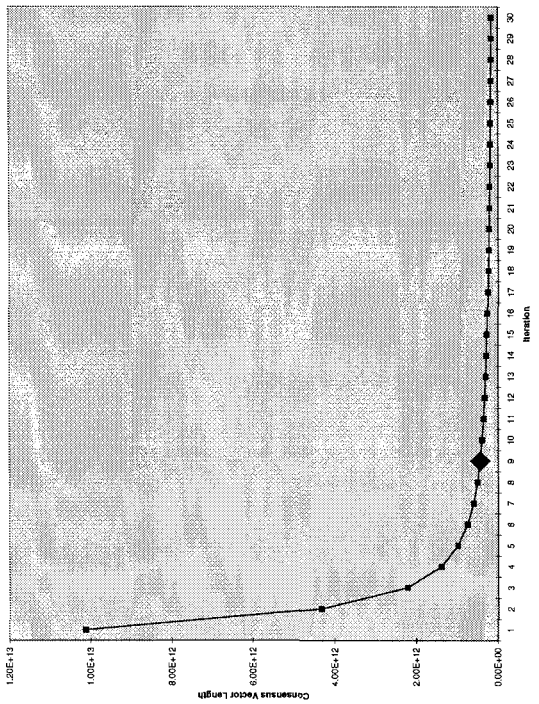

Figure A-4: Progress for Dirichlet1.mod

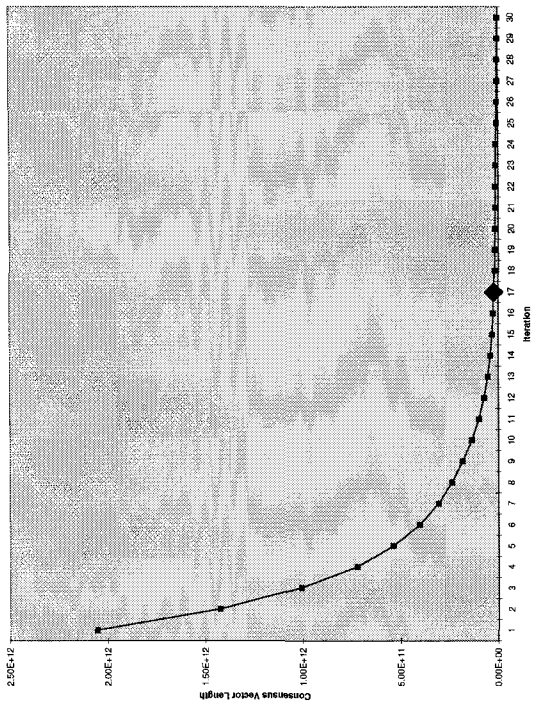

Figure A-7: Progress for Glider1.mod

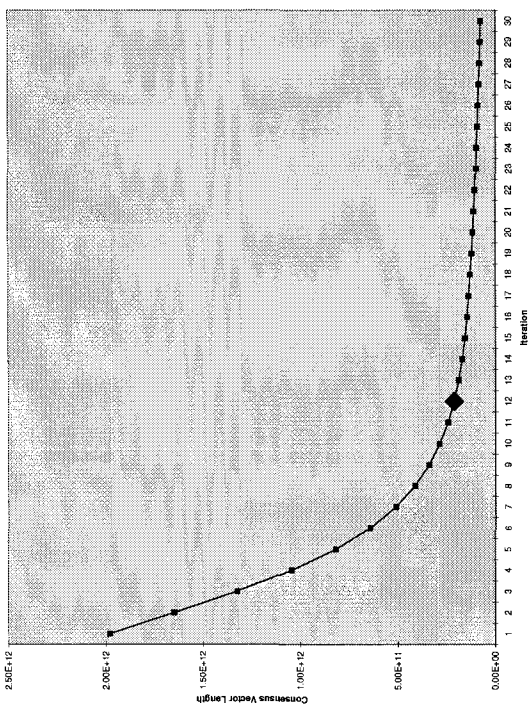

Figure A-6: Progress for Gasoil1.mod

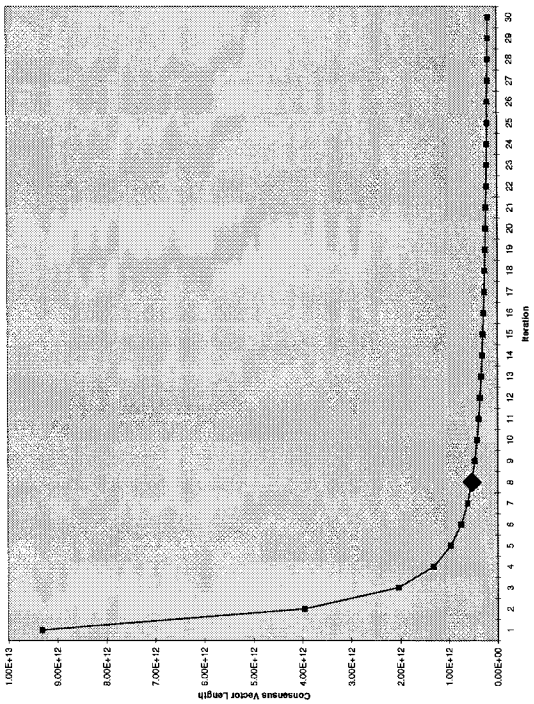

Figure A-9: Progress for Lane_emden1.mod

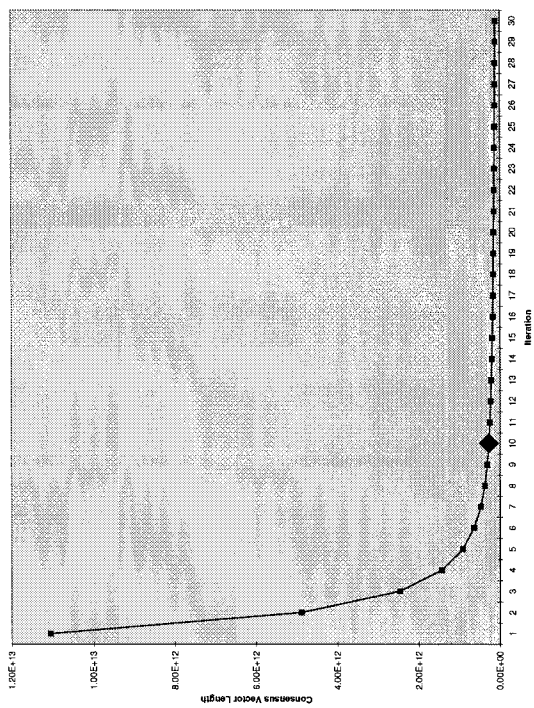

Figure A-8: Progress for Henon1.mod

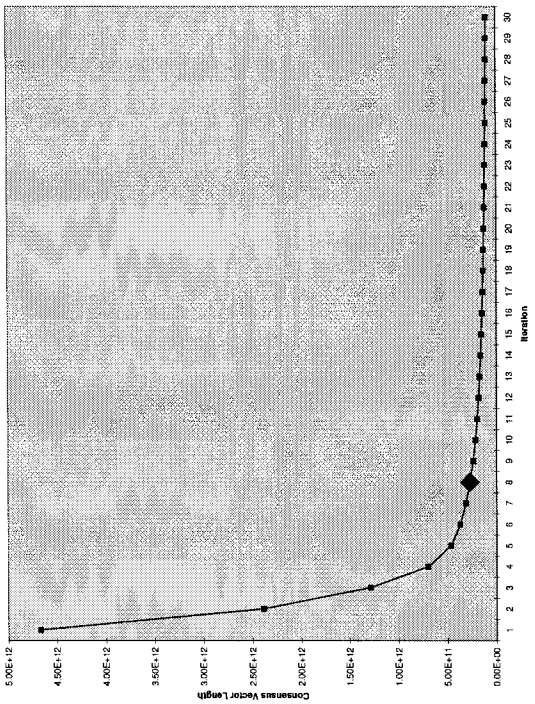

Figure A-11: Progress for Methanol1.mod

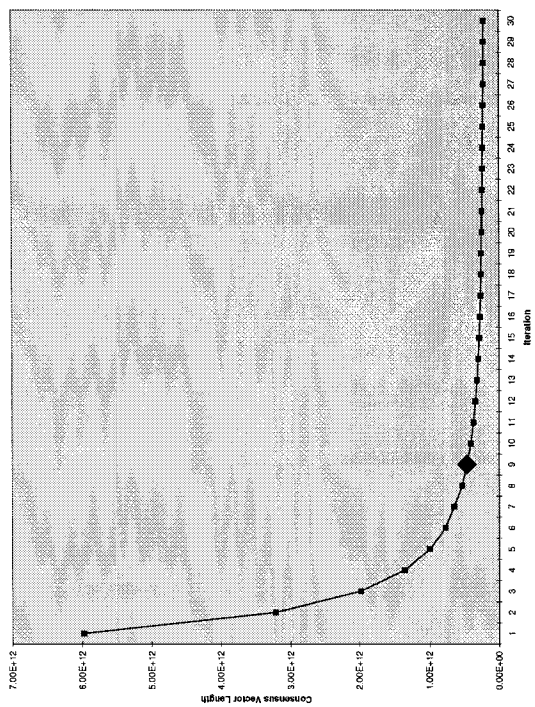

Figure A-10: Progress for Marine1.mod

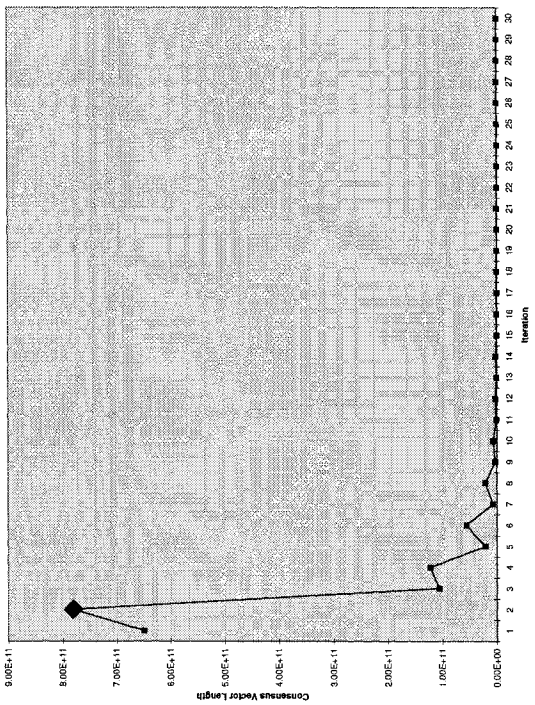

Figure A-13: Progress for Polygon1.mod

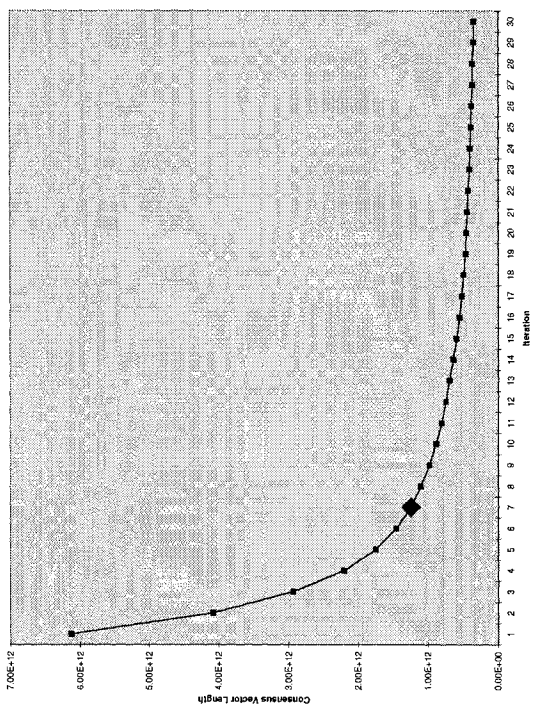

Figure A-12: Progress for Pinene1.mod

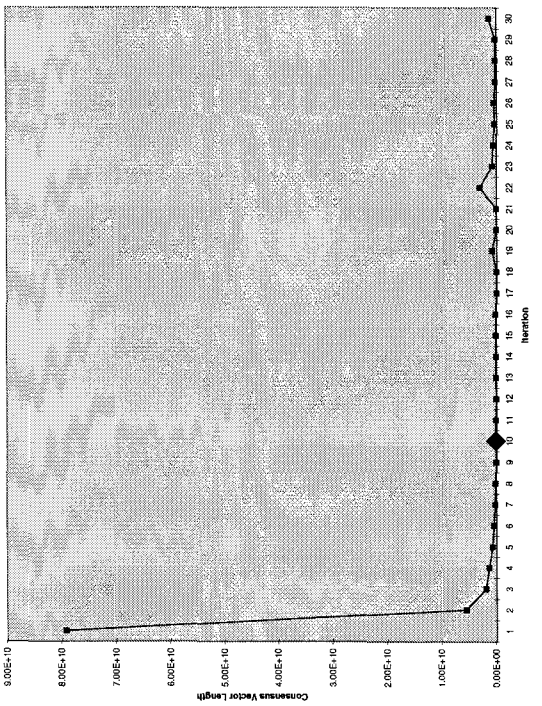

Figure A-15: Progress for Steering1.mod

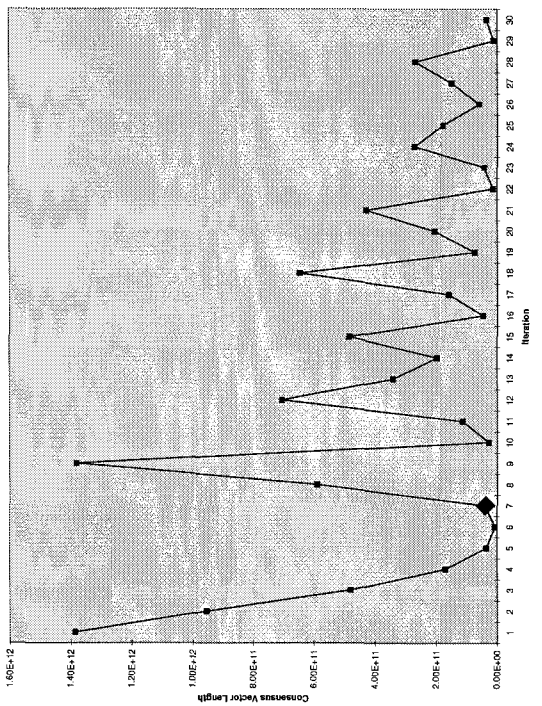

Figure A-14: Progress for Robot1.mod

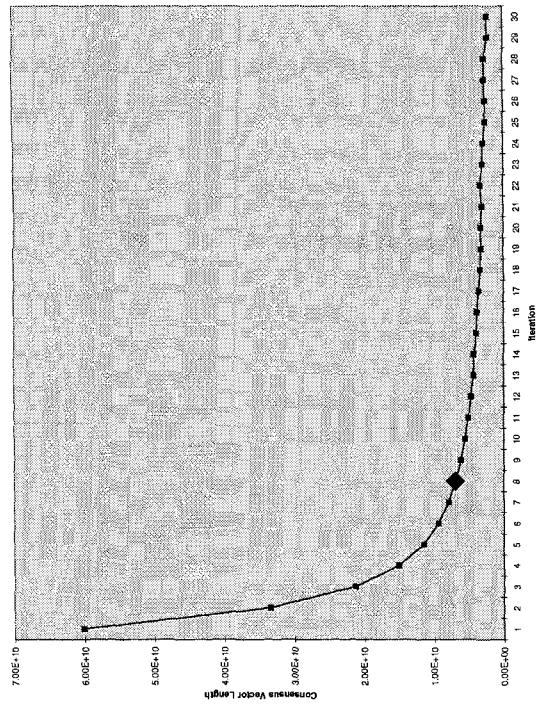

Figure A-17: Progress for Triangle1.mod

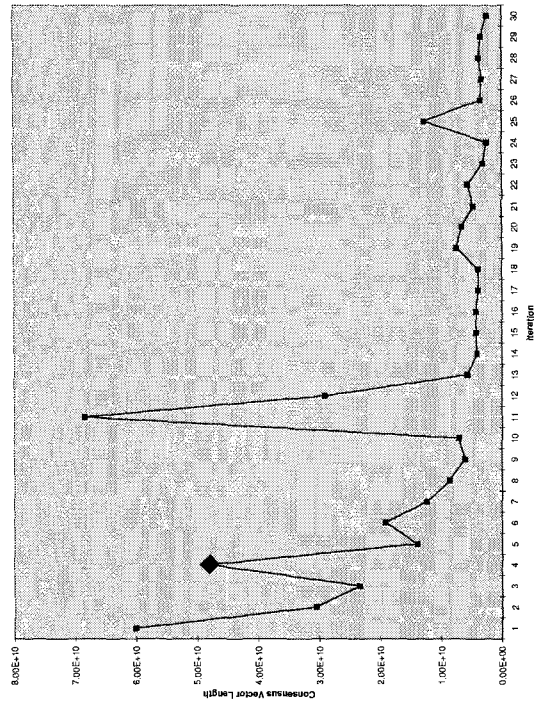

Figure A-16: Progress for Tetra1.mod