PART I: SEEKING FEASIBILITY

There are several good reasons for wanting to be able to reach feasibility quickly in a mathematical program. Some solution methods are unable to proceed to optimality without first reaching a feasible solution (most commonly for nonlinear programs). Overall solution speed is increased in some algorithms if a feasible solution is available, e.g. branch and bound solutions of mixed-integer linear programs (MIP) models can be much faster if a feasible incumbent solution is available early to help prune the subsequent tree. For many models, a feasible solution is all that is required (e.g. in scheduling applications). Finally, many methods for analyzing infeasibility require the repeated solution of subsets of the model constraints. Such methods are greatly speeded if the feasibility status of sets of constraints can be decided quickly; reaching feasibility rapidly is very helpful in this effort.

As we will see in Chapters 2–5, there is a wide variety of algorithms for seeking feasibility for all model forms. Most recently, there has been a great deal of progress in algorithms for reaching feasible solutions quickly for nonlinear programs and for mixed-integer linear programs.

A number of researchers (Mitchell et al. 1992, Mammen and Hogg 1997, Conrad et al. 2007) have noticed that the difficulty of determining feasibility status is directly related to how tightly the problem is constrained. They have observed a typical "easy-hard-easy" pattern as the model moves from lightly constrained at one extreme (in which case a feasible solution is easy to find) to tightly constrained at the other extreme (in which case it is easy to determine that the model is infeasible). The hardest feasibility problems, on average, are those in the middle range where the model is neither lightly nor tightly constrained. In the middle range, a great deal of search effort may be required to arrive at a definite determination of the feasibility status of the set of constraints. This pattern holds for many problems, including the classic satisfiability problem, graph colouring, constraint satisfaction programs, the connection subgraph problem, etc.

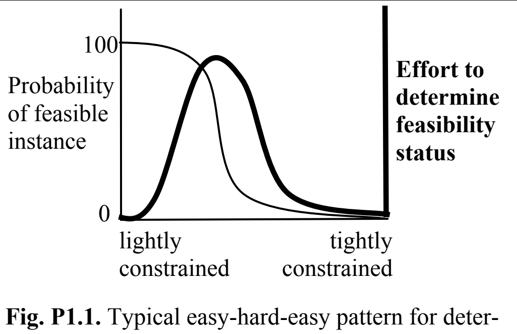

Fig. P1.1. Typical easy-hard-easy pattern for determination of feasibility status

The transition from mostly feasible instances of models to mostly infeasible instances is generally a rather sharp "phase transition". A typical diagram of the phase transition and the average computing effort to solve an instance is shown in Fig. P1.1. The computational effort to determine the feasibility status typically peaks in the region of the phase transition.

Surprisingly, Conrad et al. (2007) also observe in their experiments that proving the infeasibility of infeasible instances can be much harder than proving optimality in the computationally difficult part of the problem space for their particular connection subgraph application. This is likely a side effect of the fact that finding a single feasible solution proves feasibility, but to prove infeasibility you need to investigate all possible feasible regions. In most combinatorial problems this usually necessitates some kind of full expansion of the search tree, e.g. the full expansion of the branch and bound tree for a MIP to show that all leaf nodes are infeasible. In contrast, a single feasible leaf proves that a feasible solution exists and also helps to prune the subsequent tree search for the optimum solution by providing a bound on the optimum solution.

Model Reformulation

It is natural to work directly with the original model as supplied by the modeler. However this may not be the best form of the model for the purposes of seeking either feasibility or optimality. A given constraint can sometimes be algebraically manipulated so that it has the same mathematical properties (in the sense of the combinations of variable values that satisfy it) but has much better characteristics for use in algorithms for seeking a feasible or optimal point. This is true for all model forms, but is especially true for the more difficult NLP and mixed-integer NLP forms. We assume throughout Part I that the model has been put into the most suitable form for feasibility-seeking before the listed algorithms are applied. We next describe some reformulations that render constraints easier to satisfy.

Amarger et al. (1992) developed a system specifically for algebraically reformulating nonlinear models so that they are easier to solve. Their REFORM system preprocesses models formulated in the GAMS algebraic modeling language (Rosenthal 2007) to produce reformulated GAMS code that has better solution properties. The reformulations normally have one of these effects:

- Avoid functions that may be undefined,
- Reduce the degree of nonlinearity,
- Convexify a nonlinear model, or
- Improve scaling of variables and constraints.

Undefined functions caused by division by zero are avoided by multiplying the constraint through by the denominator. For example, $x/(y-1) \leq 2$ becomes $x-2y \leq -2$, which removes the undefined behaviour at $y=1$, and simultaneously converts the nonlinear constraint into a simpler linear form.

Undefined functions caused by logarithms and exponential exponents of non-positive values are avoided by exponential transformations. Amarger et al. (1992) provide this example: $\log(x/(y-z)) \leq d$ is transformed into $x-(y-z)\,e^{d} \leq 0$. Not only is the undefined behaviour avoided, but the transformed constraint is linear when d is fixed. The constraint $y - z \geq 0$ can also be added to the model if needed.

As seen above, nonlinearity can also be removed or reduced by multiplying through by the denominator. Consider the frequently-occurring nonlinear ratio of two functions such as $g_1(\boldsymbol{x})/g_2(\boldsymbol{x}) \leq b$. This is reformulated as $g_1(\boldsymbol{x}) - b{\cdot}g_2(\boldsymbol{x}) \leq 0$ by this tactic, which is linear if both $g_1(\boldsymbol{x})$ and $g_1(\boldsymbol{x})$ are linear. In addition, approximations that are less nonlinear are known for various commonly used functions, and these can be substituted where appropriate. Finally, some nonlinear terms can be substituted out. For example, if the variable x only ever appears in the term x^3, even if this term appears in several different places, then x^3 can be replaced everywhere by another variable e.g. w. The value of x can be recovered afterwards, when the value of w has been fixed, by solving the expression $x = w^{1/3}$. This idea extends to longer terms as well, especially if they occur frequently.

Convexifying the constraints of a nonlinear model greatly improves the ability of nonlinear solvers to find a feasible point. *Posynomial functions* (these have terms that are products of nonnegative variables with arbitrary exponents and positive coefficients) are easy to convexify by exponential transformations. For example, using the transformations $x = e^X$, $y = e^Y$, and $z = e^Z$, the nonconvex posynomial function $x^{0.6}/y + 1/z \leq b$ is transformed to the convex function $e^{(0.6X-Y)} + e^{-Z} \leq b$. Using exponential functions can make even non-posynomial functions convex, for example $\log(x - a_1) + \log(y - a_2) \leq b$ is convexified to $X + Y \leq b$ using the transformations $(x - a_1) = e^X$ and $(y - a_2) = e^Y$; this forms a convex feasible region if x and y appear only in this function, but the system is nonconvex if the transformation equalities must be included in the model because x and y appear elsewhere. In that case, the system is convex if the transformation equalities can be relaxed to $(x - a_1) \geq e^X$ and $(y - a_2) \geq e^Y$ (examples from Amarger et al. (1992)).

Amarger et al. (1992) also point out the importance of proper model *scaling*. The general idea is to avoid having variable or constraint body values of widely differing scales (e.g. x_1 has a range of 0.00001 to 0.00005 while x_2 has a range of 100,000 to 500,000). Keeping the variable and constraint values in the same general range avoids many numerical problems and improves the ability of solvers to reach both feasibility and optimality. Scaling is normally accomplished by multiplying variables and constraints by constant factors so that their ranges are approximately equalized.

Tightening of the variable bounds is also very helpful, especially for nonlinear models, a topic we will return to in Secs. 5.6.1. and 6.1.1. Nonlinear solvers are much more likely to reach a feasible point if started near the feasible region, and tightening the variable bounds improves the chances of finding a good initial point.

Special rules are also available to reformulate models containing binary or integer variables which result in simpler models for which it is faster to find an integer-feasible point. If there are simple bounds on variables such as $x \leq by$ where x is a continuous variable, y is binary, and b is a constant, then b can be reduced as tighter bounds on x are deduced (Amarger et al. 1992). Reduction rules are available for constraints composed entirely of binary variables, and of the form

$$\sum_{N^+} a_j y_j - \sum_{N^-} a_k y_k \le b$$

where N^+ and N^- are disjoint sets of variable indices, and all of a_j and a_k are strictly positive constants. Crowder et al. (1983) show that the following deductions can be made:

- If $-\sum_{N^-} a_k > b$ then the constraint cannot be satisfied and the model is infeasible.
- If $\sum_{N^+} a_j \le b$ then the constraint is always satisfied and can be removed from the model.
- If $a_j > b + \sum_{N^-} a_k$ then y_j can never be equal to 1, and hence can be fixed to 0.
- If $a_{k'} > b + \sum_{N^-} a_k$ where $a_{k'} \in N^-$, then $y_{k'}$ must be equal to 1 and can be fixed to that value.

The bounds on integer variables can also be tightened as a by-product of the tightening of continuous variables. For example if an integer value has an upper bound of 7, but treating it as continuous during bound tightening shows a tightened continuous bound of 6.8, then the integer bound can be reset to 6.

Amarger et al. (1992) give several examples of models in which no feasible solution can be found until the model is reformulated.