Global Optimization in the COCONUT project

Hermann Schichl*

Institut für Mathematik der Universität Wien
Strudlhofg. 4
A–1090 Wien
Hermann.Schichl@esi.ac.at

Abstract. In this article, a solver platform for global optimization is presented, as it is developed in the COCONUT project. After a short introduction, a short description is given of the basic algorithmic concept and of all relevant components, the strategy engine, inferenence engines, and the remaining modules. A compact description of the search graph and its nodes and of the internal model representation using directed acyclic graphs (DAGs) completes the presentation.

1 Introduction

The COCONUT project [2] is aimed at the integration of the existing approaches to continuous global optimization and constraint satisfaction. It is a project funded by the Future and Emerging Technologies (FET) arm of the IST programme FET-Open scheme of the European Community (IST-2000-26063). Six academic and one industrial partner are involved:

- ILOG Inc., Paris — the industrial partner and project coordinator,
- TU Darmstadt, Germany,
- IRIN Nantes, France,
- EPFL Lausanne, Switzerland,
- University of Vienna, Austria,
- University of Louvain-la-Neuve, Belgium,
- University of Coimbra, Portugal.

The COCONUT consortium is planning to provide at the end of the project (February 2004) a modular solver environment for nonlinear global optimization problems with an open-source kernel, which

* supported by the EU project COCONUT (IST-2000-26063)

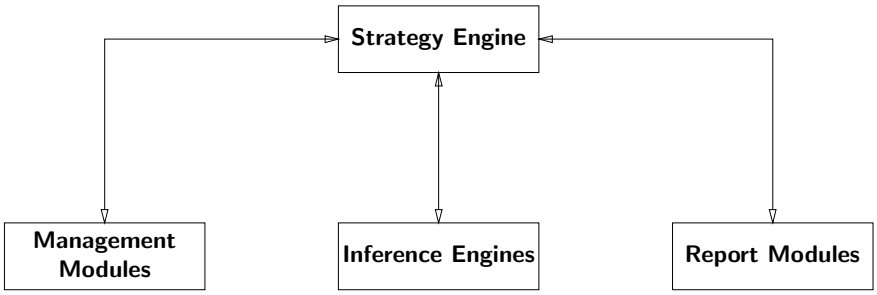

Fig. 1. Basic Scheme of the Algorith

can be expanded by commercial and open-source solver components (inference engines, see Section 5).

The application programmer's interface (API) is designed to make the development of the various module types independent of each other and independent of the internal model representation. It will be a collection of open-source `C++` classes protected by the LGPL license model, so that it could be used as part of commercial software. It uses the `FILIB++` [5] library for interval computations and the matrix template library (`MTL`) [8] for the internal representation of various matrix classes. Support for dynamic linking will relieve the user from recompilation when modules are added or removed. In addition, it is designed for distributed computing, and will probably be developed further (in the years after the end of the COCONUT project) to support parallel computing as well.

The API kernel implementation consists of more than 50.000 lines of `C++` code and a few `perl` scripts, organized into about 150 files, occupying 1.5 MB of disk space.

The algorithmic design follows the scheme depicted in Figure 1; its various parts are described in more detail in the following sections.

2 Models and the Search Graph

The solution algorithm is an advanced branch-and-bound scheme which proceeds by working on the **search graph**, a **directed acyclic graph (DAG)** of search nodes, each representing an optimization problem, a **model**. The **search nodes** come in two flavors: **full**

nodes which record the complete description of a model, and **delta nodes** which only contain the difference between the model represented by the node and its (then only) parent. All search nodes "know" in addition their relation to their ancestors. They can be splits, reductions, relaxations, or glueings. The latter turn the graph into a DAG instead of a tree, as usual in branch-and-bound algorithms. The search graph is implemented using the Vienna Graph Template Library (`VGTL`), a library following the generic programming spirit of the `C++ STL` (Standard Template Library).

A **reduction** is a problem, with additional or stronger constraints (**cuts** or **tightenings**), whose solution set can be shown to be equal to the solution set of its parent. A **relaxation** is a problem with fewer or weakened constraints, or a "weaker" objective function, whose solution set contains the solution set of its parent. Usually, relaxed problems have a simpler structure than its original. Typically *linear* or *convex* relaxations are used.

A problem is a **split** of its parent if it is one of at least two descendants and the union of the solution sets of all splits equals the solution set of their parent. Finally, a model is a **glueing** of several problems, if its solution set equals the solution sets of all the glued problems.

During the solution process some, and hopefully most, of the generated nodes will be solved, and hence become **terminal nodes**. These can be removed from the graph after their consequences (e.g., optimal solutions, ...) have been stored in the **search database**. This has the consequence that the ancestor relation of a node can change in the course of the algorithm. If, e.g., all the splits but one have become terminal nodes, this split turns into a reduction. If all children of a node become terminal, the node itself becomes terminal, and so on.

The search graph has a **focus** pointing to the model which is worked upon. This model is copied into an enhanced structure - the **work node**. A reference to this work node is passed to each inference engine activated by the strategy engine. The graph itself can be analyzed by the strategy engine using so-called **search inspectors**.

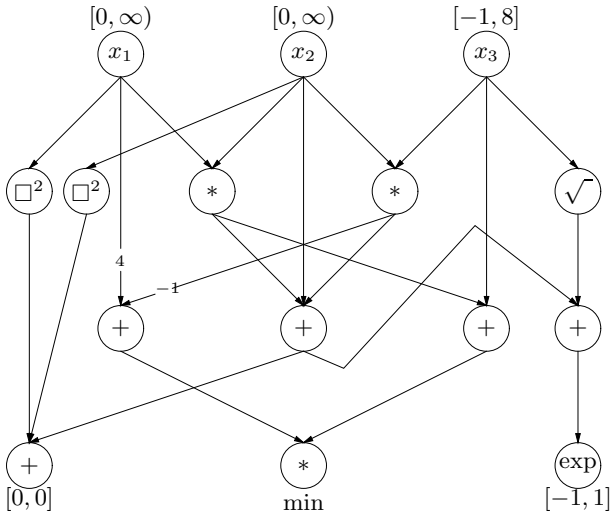

Fig. 2. DAG representation of problem (1)

3 Mathematical Representation of Problems, Directed Acyclic Graphs

The optimization problems stored in the work nodes, which are passed to the various inference engines, are kept as directed acyclic graphs (DAG), as well. This representation has big advantages; see [7] for a detailed analysis.

A complete optimization problem is always represented by a *single* DAG. The vertices of the graph represent operators similar to computational trees. Constants and variables are sources, objective and constraints are sinks of the DAG. Consider for example the optimization problem

$$\begin{aligned} \min \quad & (4x_1 - x_2x_3)(x_1x_2 + x_3) \\ \text{s.t.} \quad & x_1^2 + x_2^2 + x_1x_2 + x_2x_3 + x_2 = 0 \\ & e^{x_1x_2+x_2x_3+x_2+\sqrt{x_3}} \in [-1, 1]. \end{aligned} \tag{1}$$

This defines the DAG depicted in Figure 2.

This DAG is optimally small in the sense that it contains every subexpression of objective function and constraints only once.

Every vertex represents a function $F : \mathbb{R}^N \to \mathbb{R}$ for some N. Predefined functions include `sum`, `product`, `max`, `min`, elementary real

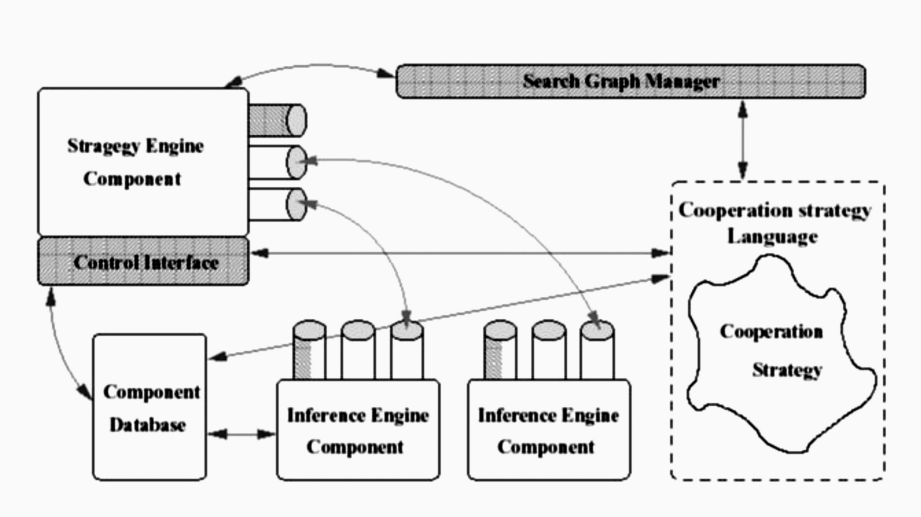

Fig. 3. The Strategy Engine Component Framework

functions (`exp`, `log`, `pow`, `sqrt`, ...), and also some discrete operators like `all_diff` and `count`.

For expression graphs (DAG or tree), special forward and backward evaluators are provided. Currently implemented are *real function values*, *function ranges*, *gradients* (real, interval), and *slopes*. In the near future evaluators for *Hessians* (real, interval) and *second order slopes* (see, e.g., [6]) will be provided, as well.

4 The Strategy Engine

The **strategy engine** is the main part of the algorithm. It makes decisions, directs the search, and invokes the various modules.

The strategy engine consists of the logic core ("**search**") which is essentially the main solution loop, special **decision makers** (very specialized inference engines, see Section 5) for determining the next action at every point in the algorithm. It calls the management modules, the report modules, and the inference engines in a sequence defined by programmable search strategies.

The engine can be programmed using a simple **strategy language**, an interpreted language based on Python. Since it is interpreted, (semi-)interactive and automatic solution processes are possible, and even debugging and single-stepping of strategies is supported. The language is object oriented, garbage collecting, and provides dynamically typed objects. These features make the system

easily extendable.

Furthermore, the strategy engine manages the search graph via the **search graph manager**, and the search database via the **database manager**.

The strategy engine uses a component framework (see Figure 3) to communicate with the inference engines. This makes it possible to launch inference engines dynamically (on need, also remote) to avoid memory overload. Since the strategy engine is itself a component, even multilevel strategies are possible.

5 Inference Engines

For the solution strategy, the most important class of modules are the **inference engines**. They provide the computational base for the algorithm, namely methods for problem structure analysis, local optimization, constraint propagation, interval analysis, linear relaxation, convex optimization, bisection, ...

Corresponding to every type of problem change, a class of inference engines is designed: **model analysis** (e.g. find convex part), **model reduction** (e.g. pruning, fathoming), **model relaxation** (e.g. linear relaxation), **model splitting** (e.g. bisection), **model glueing** (e.g. undo excessive splitting), **computing of local information** (e.g. probing, local optimization).

Inference engines calculate changes to a model that do not change the solution set. But they **never** change the model; the decision to apply the changes if they are considered useful is left to the strategy engine. Therefore, the result of an inference engine is a list of changes to the model together with a weight (the higher the weight the more important the change). Whether an advertised change is actually performed is decided by the strategy engine, and the actual change is executed by an appropriate management module. The inference engines are implemented as subclass of a single `C++` base class. In addition, there is a fixed documentation structure defined.

Several state of the art techniqnes are already provided:

- DONLP2-INTV, a general purpose nonlinear local optimizer for continuous variables ,
- STOP, a heuristic starting point generator ,

- Karush-John-Condition generator using symbolic differentiation,
- Point Verifier for verifying solution points,
- Exclusion Box generator, calculating an exclusion region around local optima [6],
- Interval constraint propagation ,
- Linear Relaxation,
- CPLEX, a wrapper for the state of the art commercial linear programming solver by ILOG,
- Basic Splitter,
- BCS, a box covering solver ,
- Convexity detection, for simple convexity analysis.

6 Management and Report Modules

Management modules are the interface between the strategy engine and the internal representation of data and modules, taking care of the management of models, resources, initialization, the search graph, the search database, ...

They are provided to make it possible to change the implementation of the search graph and the internal representation of problems without having to change all of the modules. Management modiles just perform some of the changes which have been advertised by inference engines; they **never** calculate anything.

The final class of modules, called **report modules**, produce output. Human or machine readable progress indicators, solution reports, the interface to modeling languages [4] (currently only `AMPL` [3] is supported), and the biggest part of the checkpointing is realized via report modules.

7 Conclusion

The open design of the solver architecture, and its extensibility to include both open source modules and commercial programs, was chosen in the hope that the system will be a unique platform for global optimization in the future, serving the major part of the community, bringing their members closer together.

We are happy that researchers and companies from outside the COCONUT project have already agreed to complement our efforts

in integrating the known techniques. Thus there will be in the near future **Bernstein modules** by J. Garloff and A. Smith (U. Konstanz), **verified lower bounds for convex relaxations** by Ch. Jansson (TU Hamburg-Harburg), a **GAMS reader** by the GAMS consortium [1], **Taylor arithmetic** by G. Corliss (Marquette U.), **asymptotic arithmetic** by K. Petras (U. Braunschweig), and an interface to **XPRESS**, a commercial LP-solver by Dash Optimization.

Acknowledgments

I want to thank Arnold Neumaier (University of Vienna) for his support and his advice, and Eric Monfroy (IRIN, Nantes) for the picture of the strategy engine components.

References

1. Anthony Brooke, David Kendrick, and Alexander Meeraus. *GAMS - A User's Guide (Release 2.25)*. Boyd & Fraser Publishing Company, Danvers, Massachusetts, 1992.
2. The COCONUT project home page. http://www.mat.univie.ac.at/coconut.
3. Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL — A Mathematical Programming Language*. Thomson, second edition, 2003.
4. Josef Kallrath, editor. *Modeling Languages in Mathematical Optimization*. Kluwer Academic Publishers, Boston Dordrecht London, 2003.
5. M. Lerch, G. Tischler, and J. Wolff von Gudenberg. *filib++–Interval Library, Specification and Reference Manual*. Informatik, Universität Würzburg, techn. report 279 edition, August 2001.
6. Hermann Schichl and Arnold Neumaier. Exclusion regions for systems of equations. *SIAM J. Num. Analysis*, 2003. to appear.
7. Hermann Schichl and Arnold Neumaier. Interval analysis on directed acyclic graphs for global optimization, 2003. Preprint.
8. J. Siek, A. Lumsdaine, and L.-Q. Lee. Generic programming for high performance numerical linear algebra. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1999.