Modeling and Solving Nontraditional Optimization Problems
Session 4a: Solver Interfaces

Robert Fourer
Industrial Engineering & Management Sciences
Northwestern University
AMPL Optimization LLC
4er@northwestern.edu — 4er@ampl.com

Chiang Mai University International Conference
Workshop
Chiang Mai, Thailand — 4-5 January 2011
Session 4a: **Solver Interfaces**

**Focus**
- Hooking non-traditional solvers to AMPL

**Topics**
- Needs of traditional vs. non-traditional solvers
- Building an interface to a constraint programming solver
  - walking the expression tree
  - the `numberof` operator
  - variables in subscripts
- Global nonlinear solver choice
Needs of Traditional Solvers

Linear / quadratic

- Coefficients
- Constraint constants

Nonlinear

- Function evaluations
- Derivative evaluations

... at points generated by solver
AMPL’s Open Interface

AMPL/solver interface library

Freely downloadable from
www.netlib.org/ampl/solvers/ or
netlib.sandia.gov/ampl/solvers/

“Hooking Your Solver to AMPL”

Instructions for writing a solver driver, at
www.ampl.com/ampl/hooking.html

Drivers for over 20 solvers

Source code for many in netlib
ampl/solvers/lancelot/... ampl/solvers/minos/...
ampl/solvers/lpsolve/... ampl/solvers/path/...

Packaged with commercial solvers
CONOPT, CPLEX, Gurobi, MINOS, Xpress-MP,...
Needs of Nontraditional Solvers

Global nonlinear optimization

- Complete function descriptions

Constraint programming

- Extended operators, expressions, variables
- Complete descriptions of constraint expressions
Nontraditional

Global Optimization

Needs

Function and gradient values (LGO, TUNNEL)
Complete descriptions of all expressions

Representations

Codelist of 4-tuples (GlobSol)
Compact, flexible NOP format (GLOPT)
Internal data structure created by C++ calls (Numerica)
Nontraditional

Constraint Programming

Needs

Complete descriptions of all constraint expressions

Extensions

Operators on constraints
New aggregate operators
Generalized indexing: variables in subscripts
New types of variables: object-valued, set-valued

Representations

Internal data structure created by C++ calls
(ILOG Solver, CHIP?)
Hooking Nontraditional Solvers to AMPL

Walking the expression tree

C++ driver code for constraints
Recursive tree-walk function
Tree-walk cases

Translating variables in subscripts

Overall design
Location, assignment, and sequencing examples
Motivation

Convey objective and constraint expressions to a global or constraint solver

Implementation

More types of expression nodes
Constraint nodes

Recursive walk of AMPL’s expression tree to build the solver’s data structures . . .
Hooking

“Range” Constraints

Forms

\[ \text{num-expr} = \text{num-expr} \]
\[ \text{num-expr} \leq \text{num-expr} \text{ const} \leq \text{num-expr} \leq \text{const} \]
\[ \text{num-expr} \geq \text{num-expr} \text{ const} \geq \text{num-expr} \geq \text{const} \]

Representation

- Expression tree for nonlinear part
- List of coefficients for linear part
- Lower & upper bounds on sum of linear & nonlinear parts
Hooking

Logical Constraints

**Forms**

\[
\begin{align*}
\text{constraint} & \quad \text{and} \quad \text{constraint} \\
\text{constraint} & \quad \text{or} \quad \text{constraint} \\
\text{not} \quad \text{constraint} &
\end{align*}
\]

\[
\begin{align*}
(X[1] = 0 \quad \text{and} \quad X[2] = 0) & \quad \text{or} \quad X[1] + X[2] \geq 100
\end{align*}
\]

**Representation**

Expression tree for entire constraint

Constraint nodes that point to constraint nodes

Constraint nodes that point to expression nodes
**Hooking**

**Counting Constraints**

**Forms**

\[
\begin{align*}
\text{count} & \{ \text{indexing} \} (\text{constraint-list}) \\
\text{atmost} \ num-expr & \{ \text{indexing} \} (\text{constraint-list}) \\
\text{atleast} \ num-expr & \{ \text{indexing} \} (\text{constraint-list}) \\
\text{exactly} \ num-expr & \{ \text{indexing} \} (\text{constraint-list})
\end{align*}
\]

**Representation**

\[
\begin{align*}
\text{count: expression node} & \rightarrow \text{multiple constraint nodes} \\
\text{atmost, atleast, exactly: constraint node} & \rightarrow \text{one expression node & multiple constraint nodes}
\end{align*}
\]
Hooking

Example

ILOG Concert code for constraints

IloNumVarArray X(env, n_var);
for (j = 0; j < n_var; j++)
    X[j] = IloNumVar(env, loVarBnd[j], upVarBnd[j]);

IloRangeArray Con(env, n_con);
for (i = 0; i < n_con; i++) {
    IloExpr conExpr(env);
    if (i < nlc)
        conExpr += build_expr (con_de[i].e);
    for (cg = Cgrad[i]; cg; cg = cg->next)
        conExpr += (cg -> coef) * X[cg -> varno];
    Con[i] = IloRange (loConBnd[i] <= conExpr <= upConBnd[i]);
}

IloConstraintArray LCon(env, n_lcon);
for (i = 0; i < n_lcon; i++) {
    LCon[i] = build_constr (lcon_de[i].e);
}
Hooking

Example (cont’d)

Tree walk function for expressions

```c
IloExpr build_expr (expr *e) {
    efunc *op;
    expr **ep;
    IloInt opnum;
    IloExpr partSum;
    op = e->op;
    opnum = Intcast op;
    switch(opnum) {
        .......
    }
}
```
Hooking

Example (cont’d)

Tree walk function for constraints

```c
IloConstraint build_constr (expr *e)
{
    efunc *op;
    expr **ep;
    IloInt opnum;

    op = e->op;
    opnum = Intcast op;

    switch(opnum) {
        ........
    }
}
```
Hooking

Example (cont’d)

Tree-walk cases for expression nodes

switch(opnum) {
    case PLUS_opno:
        return build_expr (e->L.e) + build_expr (e->R.e);

    case SUMLIST_opno:
        partSum = IloExpr(env);
        for (ep = e->L.ep; ep < e->R.ep; *ep++)
            partSum += build_expr (*ep);
        return partSum;

    case LOG_opno:
        return IloLog (build_expr (e->L.e));

    case CONST_opno:
        return IloNumVar (env, e->dL, e->dL);

    case VAR_opno:
        return X[e->a];

    ........
}

Robert Fourer, Modeling & Solving Nontraditional Optimization Problems
Session 4a: Solver Interfaces — Chiang Mai, 4-5 January 2011
Hooking

Example (cont’d)

Tree-walk cases for constraint nodes

```c
switch(opnum) {
  case OR_opno:
    return build_constr (e->L.e) || build_constr (e->R.e);
  case AND_opno:
    return build_constr (e->L.e) && build_constr (e->R.e);
  case LE_opno:
    return build_expr (e->L.e) <= build_expr (e->R.e);
  case EQ_opno:
    return build_expr (e->L.e) == build_expr (e->R.e);
  .......
}
```
Hooking

Example (cont’d)

Tree-walk cases for “count” operators

```
switch(opnum) {
    case OPCOUNT_opno:
        partSum = IloExpr(env);
        for (ep = e->L.ep; ep < e->R.ep; *ep++)
            partSum += Build_Constr (*ep);
        return partSum;
    .......
}
```

```
switch(opnum) {
    case ATMOST_opno:
        build_expr (e->L.e) >= build_expr (e->R.e);
    case ATLEAST_opno:
        build_expr (e->L.e) <= build_expr (e->R.e);
    .......
}
```

... right op is a “count” expression
Structure Constraints

Number-Of

Machine scheduling with capacities

subject to AssignCapJobs {i in 1..nMachines}:

numberof i in ({j in 1..nJobs} MachineForJob[j]) <= cap[i];

Treatment as “structure” constraint

Collect all numberof expressions having same expression-list

Handle them jointly in search for solution

... provided no variables in expressions following numberof
Structure Constraints

Number-Of Operator

Form

\text{numberof} \ target-expr \ \text{in} \ (\text{expression-list})

Simple tree-walk case

\begin{verbatim}
switch(opnum) {
    // build_expr
    ........
    case NUMBEROF_opno:
        ep = e->L.ep;
        targetExpr = build_expr (*ep);
        partSum = IloExpr(env);
        for (*ep++; ep < e->R.ep; *ep++)
            partSum += (build_expr (*ep) == targetExpr);
        return partSum;
\end{verbatim}

\ldots but doesn't process as a single structure constraint
Structure Constraints

Building a Number-Of Constraint

Extended tree-walk case

```c
switch(opnum) { // build_expr
    ........
    case NUMBEROF_opno:
        ep = e->L.ep;
        if ((int)*ep->op == CONST_opno) /* target is a constant */
            return build_numberof (e);
        else { /* process individually as before */ }
```

Steps in build_numberof routine

Check whether this expression-list has been seen before
Keep track of target-exprs encountered for each expression-list

... generate IloDistribute calls
after all AMPL constraints have been processed
Hooking

Variables in Subscripts

**Overall design (C++ interface)**

Driver sets up single array \( x \) of variables

New node type represents

subscripting by expression containing variables

Solver accepts \( x[expr-involving-vars] \)

by overloading of the subscripting operator

**Complications**

Conversion of subscript values to fit \( x \) array

Variables in subscripts of *parameters*

\[ \ldots \textit{avoid high-level model info in driver code} \]
Parametric Hooking Example 1

Location

\begin{align}
\text{param } & mCL \text{ integer } > 0; \\
\text{param } & nWH \text{ integer } > 0; \\
\text{param } & \text{srvCost } \{1..mCL, 1..nWH\} > 0; \\
\text{param } & \text{bdgCost } > 0; \\
\text{var } & \text{Serve } \{1..mCL\} \text{ integer } \geq 1, \leq nWH; \\
\text{var } & \text{Open } \{1..nWH\} \text{ binary;}
\end{align}

\textbf{minimize} \text{TotalCost:}
\begin{align}
\text{sum } \{i \text{ in } 1..mCL\} \text{ srvCost}[i,\text{Serve}[i]] + \\
\text{bdgCost } \times \text{sum } \{j \text{ in } 1..nWH\} \text{ Open}[j];
\end{align}

\textbf{subject to} \text{OpenDefn } \{i \text{ in } 1..mCL\}:
\begin{align}
\text{Open[Serve[i]] } & = 1;
\end{align}

Convert \text{Open[Serve[i]]} to \text{X[offset0 + X[offsetS+i]]}
Hooking

Subscript Example 2

Assignment

\[
\begin{align*}
\text{param } & \quad n \text{ integer } > 0; \\
\text{set } & \quad \text{JOBS} := 1..n; \\
\text{set } & \quad \text{MACHINES} := 1..n; \\
\text{param } & \quad \text{cost} \{\text{JOBS},\text{MACHINES}\} > 0; \\
\text{var } & \quad \text{MachineForJob} \{\text{JOBS}\} \text{ integer } \geq 1, \leq n; \\
\text{minimize } & \quad \text{TotalCost:} \\
& \quad \sum \{j \text{ in } \text{JOBS}, k \text{ in } \text{MACHINES}\} \text{ cost}[j,\text{MachineForJob}[j]]; \\
\text{subj to } & \quad \text{OneJobPerMachine:} \\
& \quad \text{alldiff} \{j \text{ in } \text{JOBS}\} \text{ MachineForJob}[j];
\end{align*}
\]

Convert \text{cost}[j,\text{MachineForJob}[j]] to

\[P[\text{offsetC} + n\text{Job}*(j-1) + X[\text{offsetM}+j]]\]

where \(P\) is an array of parameter values

\[\text{... pass parameter array via .nl file}\]
\[\text{(another extension)}\]


Hooking

Subscript Example 3

Sequencing

param duePen {0..nJobs} >= 0;
param dueTime {0..nJobs} >= 0;
param classOf {0..nJobs} in 0..nClasses;
param setupCost {0..nClasses,1..nClasses};
var JobForSlot {k in 0..nSlots} in 0..nJobs;
var ComplTime {j in 0..nJobs};

minimize CostPlusPenalty:

sum {k in 1..nSlots} setupCost[classOf[JobForSlot[k-1]],classOf[JobForSlot[k]]] + 
sum {j in 1..nJobs} duePen[j] * (dueTime[j] - ComplTime[j]);

Tree for subscript of setupCost [...] contains 2 more variable-in-subscript nodes
Assignment

set JOBS;
set MACHINES;
set ABLE within {JOBS,MACHINES};

param cost {ABLE} > 0;

var MachineForJob {JOBS} in MACHINES;

minimize TotalCost:
    sum {j in JOBS} cost[j,MachineForJob[j]];

subj to OneJobPerMachine:
    alldiff {j in JOBS} (MachineForJob[j]);

subj to MachineCanDoJob {j in JOBS}:
    (j,MachineForJob[j]) in ABLE;

No simple rule for conversion of
    cost[j,MachineForJob[j]] to P[expr]
— can only give (job, machine, cost) table

Is MachineCanDoJob constraint necessary?
Related Writings

AMPL and Solvers

http://www.ampl.com/ampl/hooking.html
http://www.ampl.com/ampl/REFS/


Related Writings

AMPL and Constraint Programming


Global Nonlinear Solver Choice

Multi-modal error function

\[
\text{var } X \{1..2\} \geq -5, \leq 5, := \text{Uniform}(-5,5);
\]

\[
\text{minimize Error:}
(\text{X}[1] - \sin(2*X[1] + 3*X[2]) - \cos(3*X[1] - 5*X[2]))^2 + \\
(\text{X}[2] - \sin(X[1] - 2*\text{X}[2]) + \cos(X[1] + 3*X[2]))^2;
\]
Global Nonlinear \textit{(cont’d)}

\textbf{Classical local methods}

\begin{verbatim}
ampl: model multimodal.mod;
ampl: let \{j in 1..2\} X[j] := Uniform(-5,5);
ampl: option solver knitro;
ampl: solve;
\end{verbatim}

\textbf{KNITRO 5.0:}
LOCALLY OPTIMAL SOLUTION FOUND.
objective $3.543865e-01$; feasibility error $0.000000e+00$
9 major iterations; 11 function evaluations

\textbf{LOQO 6.07:} optimal solution (9 iterations, 10 evaluations)
primal objective $5.814508861$
\hspace{1cm} dual objective $5.814508739$

\textbf{CONOPT 3.14D:} Locally optimal; objective $1.520773908$
10 iterations; evals: $nf = 21$, $ng = 8$, $nc = 0$, $nJ = 0$, $nH = 0$, $n Hv = 5$
Global Nonlinear (cont’d)

Local search heuristic

**PSwarm**: Variables scaled by:
- scale[0]=1.000000
- scale[1]=1.000000

Delta for pattern search: 5.000000

Stopping due to single particle and tolerance

The very best
- \( p(16) = [0.1333187035, -2.0965765856] \)
- \( f(16) = 0.0318656723 \)
- maxnormv=7.40078565638427754436
- delta=0.00000953674316406250

33 iterations
283 function evaluations
32 poll steps performed
13 poll steps performed with success
33 & 283 & 32 & 13 & 0.0319

Normal exit
Global Nonlinear (cont’d)

**KNITRO’s multistart method**

```plaintext
ampl: option knitro_options 'msenable 1 ms_maxsolves 100';
ampl: solve;

KNITRO 5.2.0:
MULTISTART: Best locally optimal point is returned.
EXIT: Locally optimal solution found.

# of iterations = 597
# of CG iterations = 329
# of function evaluations = 926
# of gradient evaluations = 697
# of Hessian evaluations = 597

Total program time (secs) = 0.32733 (0.284 CPU time)
Time spent in evaluations (secs) = 0.02119

KNITRO 5.2.0: Locally optimal solution.
objective 2.3869889306092854e-21; feasibility error 0
597 major iterations; 926 function evaluations
```
Global Nonlinear (cont’d)

LGO’s global method

```ampl
ampl: model multimodal.mod;
ampl: let {j in 1..2} X[j] := Uniform(-5,5);
ampl: option solver lgo;
ampl: solve;
```

LGO: Globally Optimal Solution

Objective 7.474818358e-23
1550 function evaluations
Runtime = 0 seconds