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

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)

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

Walking the Expression Tree

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

num-expr = num-expr

num-expr <= num-expr *const <= num-expr <= const*

num-expr >= num-expr *const >= num-expr >= const*

Representation

Expression tree for nonlinear part

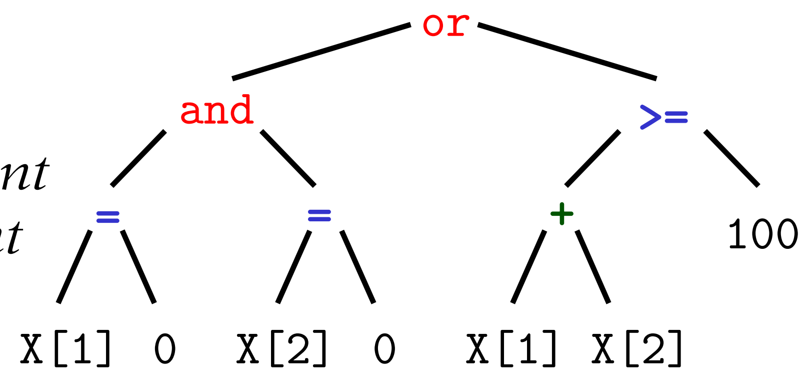
List of coefficients for linear part

Lower & upper bounds
on sum of linear & nonlinear parts

Logical Constraints

Forms

constraint `and` *constraint*
constraint `or` *constraint*
`not` *constraint*



`(X[1] = 0 and X[2] = 0) or X[1] + X[2] >= 100`

Representation

Expression tree for entire constraint

Constraint nodes that point to constraint nodes

Constraint nodes that point to expression nodes

Counting Constraints

Forms

`count` { *indexing* } (*constraint-list*)

`atmost` *num-expr* { *indexing* } (*constraint-list*)
`atleast` *num-expr* { *indexing* } (*constraint-list*)
`exactly` *num-expr* { *indexing* } (*constraint-list*)

Representation

`count`: expression node
→ multiple constraint nodes

`atmost`, `atleast`, `exactly`: constraint node
→ one expression node & multiple constraint nodes

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);
}
```

Example *(cont'd)*

Tree walk function for expressions

```
IloExpr build_expr (expr *e)

{
   efunc *op;
   expr **ep;
   IloInt opnum;
   IloExpr partSum;

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

   switch(opnum) {

      .......

   }
}
```

Example *(cont'd)*

Tree walk function for constraints

```
IloConstraint build_constr (expr *e)

{
   efunc *op;
   expr **ep;
   IloInt opnum;

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

   switch(opnum) {

      .......

   }
}
```

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];

 .......

}
```

Example *(cont'd)*

Tree-walk cases for constraint nodes

```
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);

.......

}
```

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

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`**

Number-Of Operator

Form

`numberof` *target-expr* `in` (*expression-list*)

Simple tree-walk case

```
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;
```

. . . but doesn't process as a single structure constraint

Building a Number-Of Constraint

Extended tree-walk case

```
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-expr*s encountered for each *expression-list*

. . . generate* `IloDistribute` *calls
after all AMPL constraints have been processed

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*

. . . avoid high-level model info in driver code

Subscript Example 1

Location

```
param mCL integer > 0;
param nWH integer > 0;

param srvCost {1..mCL, 1..nWH} > 0;
param bdgCost > 0;

var Serve {1..mCL} integer >= 1, <= nWH;
var Open {1..nWH} binary;

minimize TotalCost:
   sum {i in 1..mCL} srvCost[i,Serve[i]] +
   bdgcost * sum {j in 1..nWH} Open[j];

subject to OpenDefn {i in 1..mCL}:
   Open[Serve[i]] = 1;
```

Convert `Open[Serve[i]]` to `X[offsetO + X[offsetS+i]]`

Hooking

Subscript Example 2

Assignment

```
param n integer > 0;

set JOBS := 1..n;
set MACHINES := 1..n;

param cost {JOBS,MACHINES} > 0;

var MachineForJob {JOBS} integer >= 1, <= n;

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

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

Convert `cost[j,MachineForJob[j]]` to

`P[offsetC + nJob*(j-1) + X[offsetM+j]]`

where `P` is an array of parameter values

*. . . **pass parameter array via .nl file (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

Hooking

Subscript Example 4

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/

D.M. Gay, "Hooking Your Solver to AMPL." Technical report, Bell Laboratories, Murray Hill, NJ (1993; revised 1994, 1997).

R. Fourer and D.M. Gay, "Conveying Problem Structure from an Algebraic Modeling Language to Optimization Algorithms." In *Computing Tools for Modeling, Optimization and Simulation: Interfaces in Computer Science and Operations Research,* M. Laguna and J.L. González Velarde, eds., Kluwer Academic Publishers (Dordrecht, 2000).

Related Writings

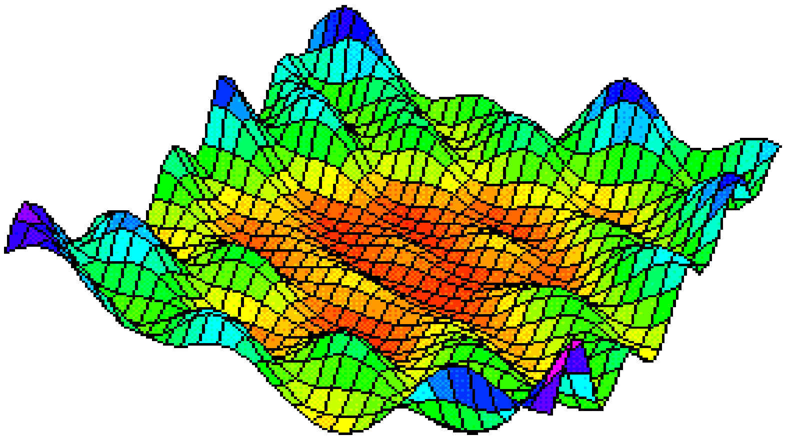
AMPL and Constraint Programming

J.J. Bisschop and R. Fourer, “New Constructs for the Description of Combinatorial Optimization Problems in Algebraic Modeling Languages.” *Computational Optimization and Applications* 6 (1996) 83–116.

R. Fourer, “Extending a General-Purpose Algebraic Modeling Language to Combinatorial Optimization: A Logic Programming Approach.” In *Advances in Computational and Stochastic Optimization, Logic Programming, and Heuristic Search: Interfaces in Computer Science and Operations Research,* D.L. Woodruff, ed., Kluwer Academic Publishers (Dordrecht, 1998).

Global Nonlinear Solver Choice

Multi-modal error function



```
var X {1..2} >= -5, <= 5, := Uniform(-5,5);

minimize Error:
   (X[1] - sin(2*X[1] + 3*X[2]) - cos(3*X[1] - 5*X[2]))^2 +
   (X[2] - sin(X[1] - 2*X[2]) + cos(X[1] + 3*X[2]))^2;
```

Global Nonlinear *(cont'd)*

Classical local methods

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

ampl: option solver knitro;
ampl: solve;

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

.......

LOQO 6.07: optimal solution (9 iterations, 10 evaluations)
primal objective 5.814508861
  dual objective 5.814508739

.......

CONOPT 3.14D: Locally optimal; objective 1.520773908
10 iterations; evals: nf = 21, ng = 8, nc = 0, nJ = 0, nH = 0, nHv = 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

```
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: 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
```