THE AMPL INTERFACE TO CONSTRAINT PROGRAMMING SOLVERS

Victor Zverovich, Robert Fourer

AMPL Optimization

The 13th INFORMS Computing Society Conference (ICS)
January 6th - 8th, 2013, Santa Fe, New Mexico, USA
WHY AMPL?

- **AMPL** is a popular algebraic modeling language:
  - used in businesses, government agencies, and academic institutions (over 100 courses in 2012)
  - large community
    (> 1,300 members in **AMPL Google Group** alone)
  - the most popular input format on **NEOS**
    (> 200,000 or 57% submissions in 2012)
- **AMPL** supports a wide range of problem types: linear, mixed integer, quadratic, second-order cone, nonlinear, complementarity problems and more.
Constraint programming (CP) allows natural formulation of many combinatorial optimization problems.

AMPL has supported CP since early 2000s when the paper Extending an Algebraic Modeling Language to Support Constraint Programming \(^1\) was published.

CP solvers connected to AMPL:
- IBM/ILOG CP Optimizer (ilogcp)
- Gecode
SUPPORTED CP CONSTRUCTS

- Logical operators: and, or, not; iterated exists, forall
- Conditional operators: if-then, if-then-else, ==>, ==> else, <==, <==>
- Counting operators: iterated count, atmost, atleast, exactly, numberof
- Pairwise operators: alldiff

AMPL SOLVER LIBRARY

AMPL Solver Library (ASL) is an open source library for connecting solvers to AMPL.

- Available from:
  - GitHub: [https://github.com/vitaut/ampl](https://github.com/vitaut/ampl)
- Includes drivers for several solvers:
  - CPLEX
  - Gurobi
  - MINOS
  - ...
AMPL SOLVER INTERFACES

- C interface:
  - described in *Hooking Your Solver to AMPL*
  - used by most solvers
- C++ interface (new):
  - a very thin wrapper around the C interface
  - type-safe, no casts needed when working with expression trees
  - easy to use, less boilerplate code
  - efficient
  - used by ilogcp and gecode
## PERFORMANCE

<table>
<thead>
<tr>
<th>Problem</th>
<th>Input+Conv Time</th>
<th>C API Time</th>
<th>C++ API Time</th>
<th>C/C++ Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign1</td>
<td>2.001</td>
<td>0.0156233</td>
<td>0.0154757</td>
<td>1.009537533</td>
</tr>
<tr>
<td>balassign1</td>
<td>2.106</td>
<td>0.086136</td>
<td>0.085996</td>
<td>1.0016279827</td>
</tr>
<tr>
<td>flowshp1</td>
<td>0.529</td>
<td>0.000594689</td>
<td>0.000598812</td>
<td>0.9931147004</td>
</tr>
<tr>
<td>flowshp2</td>
<td>0.784</td>
<td>0.000593329</td>
<td>0.00059615</td>
<td>0.9952679695</td>
</tr>
<tr>
<td>magic</td>
<td>0.558</td>
<td>0.00276411</td>
<td>0.0028042</td>
<td>0.9857035875</td>
</tr>
<tr>
<td>mapcoloring</td>
<td>0.557</td>
<td>8.6488E-007</td>
<td>8.6538E-007</td>
<td>0.9994222191</td>
</tr>
<tr>
<td>money</td>
<td>0.472</td>
<td>0.000241843</td>
<td>0.000238001</td>
<td>1.0161427893</td>
</tr>
<tr>
<td>nqueens</td>
<td>1.073</td>
<td>8.6449E-007</td>
<td>8.6593E-007</td>
<td>0.998337048</td>
</tr>
<tr>
<td>sched1</td>
<td>0.595</td>
<td>0.0172613</td>
<td>0.0174135</td>
<td>0.9912596549</td>
</tr>
<tr>
<td>sched2</td>
<td>0.684</td>
<td>0.0142709</td>
<td>0.0146368</td>
<td>0.9750013664</td>
</tr>
<tr>
<td>party1</td>
<td>33.688</td>
<td>3.88768</td>
<td>3.94516</td>
<td>0.9854302487</td>
</tr>
<tr>
<td>party2</td>
<td>126.194</td>
<td>1.20008</td>
<td>1.21847</td>
<td>0.9849073018</td>
</tr>
<tr>
<td>sudokuHard</td>
<td>0.661</td>
<td>8.6618E-007</td>
<td>8.6361E-007</td>
<td>1.0029758803</td>
</tr>
<tr>
<td>sudokuVeryEasy</td>
<td>0.726</td>
<td>8.6547E-007</td>
<td>8.6383E-007</td>
<td>1.0018985217</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.9957590574</td>
</tr>
</tbody>
</table>

Time is in milliseconds.
SOLVER WORKFLOW

1. Get a problem instance in AMPL form
2. Process solver options
3. Convert the problem from AMPL to solver form
4. Solve the problem
5. Return solution(s)

Step 3 is the most interesting especially for a CP solver, because it has to deal with expression trees. Other steps are easy to implement.
Both IBM/ILOG CP Optimizer and Gecode use C++ for their main APIs. Therefore I'll give all examples in C++ with the new interface library.

However, everything discussed here is possible to do with the C API with a bit more work.
EXPRESSION TREES

- **Expr**
  - **NumericExpr**
    - **UnaryExpr** (unary -, abs, tan, ...)
    - **BinaryExpr** (+, -, *, /, div, less, ...)
    - **VarArgExpr** (min, max)
    - **SumExpr**
    - **CountExpr**
    - **IfExpr**
    - **PiecewiseLinearTerm**
    - **NumericConstant**
    - **Variable**
    - **NumberOfExpr**
  - **LogicalExpr**
    - **LogicalConstant** (0 or 1)
    - **RelationalExpr** (<, <=, =, !=, >=, >)
    - **NotExpr** (!)
    - **LogicalCountExpr** (atleast, atmost, exactly)
    - **BinaryLogicalExpr** (||, &&, <=>)
    - **ImplicationExpr** (==) else
    - **IteratedLogicalExpr** (exists, forall)
    - **AllDiffExpr**
WORKING WITH EXPRESSIONS

- `c.value()` returns the value of a numeric or logical constant `c`.
- `e.arg()` returns an argument of a unary expression `e`.
- `e.lhs()` and `e.rhs()` return arguments of a binary expression `e`.
- `Cast<ExprType>(e)` casts expression `e` to `ExprType` if possible, otherwise returns a null expression. Fast alternative to `dynamic_cast`. 
ITERATING OVER ARGUMENTS

- Iterating over arguments of an expression e with variable number of arguments:
  
  ```cpp
  for (auto arg: e) // C++11
    // use arg
  ```

  or

  ```cpp
  for (SumExpr::iterator i = e.begin(); i != e.end(); ++i)
    // use *i
  ```

- Works with VarArgExpr, SumExpr, CountExpr, NumberOfExpr, IteratedLogicalExpr and AllDiffExpr.
TREE TRAVERSAL WITH VISITORS

ilogcp

```java
class IlogCPSolver :
    public ExprVisitor<IlogCPSolver, IloExpr, IloConstraint> { 
        public:
            // Convert logical expressions.
            // Convert numeric expressions.
    };
```

gecode

```java
class NLToGecodeConverter :
    private ExprVisitor<NLToGecodeConverter,
                        Gecode::LinExpr, Gecode::BoolExpr> { 
        // Convert logical expressions.
        // Convert numeric expressions.
        public:
            LinExpr ConvertFullExpr(NumericExpr e) { return Visit(e); }
            BoolExpr ConvertFullExpr(LogicalExpr e) { 
                // Process global constraints calling Visit(e)
                // for nested expressions.
            }
    };
```
CONVERTING NUMERIC EXPRESSIONS

IloExpr VisitNumericConstant(NumericConstant n) {
    return IloExpr(env_, n.value());
}

IloExpr VisitVariable(Variable v) {
    return vars_[v.index()];
}

IloExpr VisitPlus(BinaryExpr e) {
    return Visit(e.lhs()) + Visit(e.rhs());
}

IloExpr VisitPow(BinaryExpr e) {
    return IloPower(Visit(e.lhs()), Visit(e.rhs()));
}

IloExpr VisitSum(SumExpr e) {
    IloExpr sum(env_);
    for (auto arg : e) // C++11
        sum += Visit(arg);
    return sum;
}
CONVERTING LOGICAL EXPRESSIONS

IloConstraint VisitLogicalConstant(LogicalConstant c) {
    return IloNumVar(env_, 1, 1) == c.value();
}

IloConstraint VisitEqual(RelationalExpr e) {
    return Visit(e.lhs()) == Visit(e.rhs());
}

IloConstraint VisitGreater(RelationalExpr e) {
    return Visit(e.lhs()) > Visit(e.rhs());
}

IloConstraint VisitAnd(BinaryLogicalExpr e) {
    return Visit(e.lhs()) && Visit(e.rhs());
}

IloConstraint IloCP::VisitExists(IteratedLogicalExpr e) {
    IloOr disjunction(env_);
    for (auto arg: e) // C++11
        disjunction.add(Visit(arg));
    return disjunction;
}
HANDLING NUMBEROF

```cpp
class IlogCPSolver {
   // CreateVar is a functor that creates an IlogCP variable.
   NumberOfMap<IloIntVar, CreateVar> numberofs_;
   ...
};

IloExpr IlogCPSolver::VisitNumberOf(NumberOfExpr e) {
   NumericExpr value = e.value();
   if (NumericConstant num = Cast<NumericConstant>(value))
       return numberofs_.Add(num.value(), e);
   IloExpr sum(env_);
   IloExpr concert_value(Visit(value));
   for (Expr arg: e)
       sum += (Visit(arg) == concert_value);
   return sum;
}
```

**NumberOfMap** is a map from `NumberOf` expressions with the same argument lists to values and corresponding variables. Such expression can be converted to a single `IloDistribute` constraint.
EXPRESSION VISITOR

Copy the great architectures.
-- Edward Tufte

- Inspired by the AST visitor from the Clang compiler frontend
- Visitor design pattern with static instead of dynamic polymorphism
- Uses curiously recurring template pattern
- Very efficient: no virtual function calls, Visit* functions can be inlined
TWO-LEVEL CONVERSION

1. Top level - global constraints such as alldiff and possible optimizations for the case when expression value is not used
   - ilogcp: no extra work is necessary, the Concert interface does necessary processing
   - gecode: manual handling of alldiff in ConvertFullExpr

2. General case for nested expressions
   Example:
   \[
   \text{s.t. c: alldiff \{j in 1..n\} Row[j]+j);}
   \]
   alldiff (...) - top level expression
   Row[j] + j - subexpression
SUPPORTING MULTIPLE SOLVERS

- Separate hierarchies for logical and numeric expressions (ilogcp and gecode) are handled easily.
- Possible to deal with more complex expression hierarchies, but with more efforts.
- Not necessary to convert all expressions, solver will report an error when unhandled expression is encountered and exit gracefully. For example, gecode doesn't support many nonlinear expressions.
# Place n queens on an n by n board
# so that no two queens can attack
# each other (nqueens.mod).

param n integer > 0;
var Row {1..n} integer >= 1 <= n;

subj to c1: alldiff ({j in 1..n} Row[j]);
subj to c2: alldiff ({j in 1..n} Row[j]+j);
subj to c3: alldiff ({j in 1..n} Row[j]-j);

More examples available at
http://www.ampl.com/NEW/LOGIC/examples.html
EXAMPLE - ILOGCP

ampl: model nqueens.mod;
ampl: let n := 20;
ampl: option solver ilogcp;
ampl: solve;
ilogcp 12.4.0: feasible solution
2898 choice points, 1286 fails
ampl: display Row;
Row [*] :=
  1 14
  2 16
  3 8
  4 6
  5 15
  6 3
  ...
19 10
20 5
;
ampl: model nqueens.mod;
ampl: let n := 20;
ampl: option solver gecode;
ampl: solve;

Ampl code 3.7.3: feasible solution
220 nodes, 105 fails

ampl: display Row;
Row [*] :=
  1  1
  2  3
  3  5
  4  7
  5 14
  6 10
  ...
  19 15
  20 13

;
LINKS

- GitHub repository: https://github.com/vitaut/ampl
- C++ Solver API: solvers/util
- Gecode AMPL solver: solvers/gecode
- IlogCP AMPL solver: solvers/ilogcp