argmat-mpg: Modeling and Programming Argumentation Problems with Gecode*

Fuan Pu, Hang Ya, and Guiming Luo

School of Software, Tsinghua University, Beijing, China
`Pu.Fuan@gmail.com`, `yah16@mails.tsinghua.edu.cn`, `gluo@tsinghua.edu.cn`

Abstract. `argmat-mpg` is a software library for utilizing Gecode to solve argumentation reasoning tasks. This description presents the approaches to model and program argumentation problems as Boolean constraint satisfaction problems based on Boolean matrix algebra, and a variable selection strategy to speed up the calculation process.

1 Modeling and programming with Gecode

There are numerous works showing that each Dung's argumentation semantics [1] can be viewed as a Boolean Constraint Satisfaction Problem (BCSP). In this paper, we mainly concern on the Boolean Matrix Algebra (BMA) based constraint models proposed in [2], since these constraint models have an intuitive connection with argumentation semantics, and are easy to use and understand.

Gecode is a set of C++ libraries[1]. It provides a generic constraint development environment for constraint-based systems and applications, in which the models (BCSP's in this context) are C++ programs that must be compiled with Gecode libraries and excuted to get a solution. Gecode can support almost any constraint over Booleans. By using Gecode, we not only program the BMA-based constraint models in [2], but also program the CNF conversations generated from these constraint models (see the description of our other solver `argmat-sat` also submitted to ICCMA-2017). Figure 1 shows the programming model of the CO semantics using the BMA-based encoding [CO2]. This model is programmed using the MiniModel, a package of Gecode, which provides a more natural syntax to construct expressions and constraints using the standard C++ operators (see Fig. 2). Using these syntax, the function **neutrality** (in line 4) implements the multiplication of a Boolean attack matrix and a Boolean (variable or expression) vector, and the function **encode_CO_bma** (in line 19) gives the final programming model. Clearly, this model uses $|\mathcal{X}|$ Boolean variables, in which $\mathcal{X}$ is the argument set in the argumentation framework $\Delta = \langle \mathcal{X}, \mathcal{R} \rangle$.

Figure 3 presents the programming model for the complete semantics using the CNF encoding of [CO1] with $2 \times |\mathcal{X}|$ variables (see the clauses for argument x_i in Eqn. (1)). However, this model is not programmed by MiniModel, but by another package Int of

* This work was supported by the Fund NSFC61572279.

[1] `http://www.gecode.org/`

```
// The implementation of the neutrality function N(x)
// The parameter vars can be a vector of Boolean variables or expressions
template<class VarVector> vector<BoolExpr>
neutrality(const VecVecType& attackerVec, const VarVector& vars) {
    vector<BoolExpr> vec_bexpr;
    BoolVar bv0(*this, 0, 0);
    for (const VecType& attackers: attackerVec) {
        BoolExpr sum_expr(bv0);
        for (auto j: attackers) {
            sum_expr = (sum_expr || vars[j]);  // Disjunction of all attackers.
        }
        vec_bexpr.emplace_back(!sum_expr);
    }
    return vec_bexpr;  // return a vector of Boolean expressions
}

// The programming model of CO semantics using x = N(x) * N(N(x))
void encode_CO_bma() {
    new_vars(m_daf.getNumberOfArguments()); // Create argNum variables

    // Get attacker list of each argument
    VecVecType attackerVec = m_daf.getAttackersVector();

    vector<BoolExpr> bva1 = neutrality(attackerVec, X);      // N(x)
    vector<BoolExpr> bva2 = neutrality(attackerVec, bva1);  // N(N(x))
    for (auto i = m_daf.getNumberOfArguments(); i--; ) {
        rel(*this, X[i] == (bva1[i] && bva2[i]));   // Post the constraints
    }
}
```

Fig. 1. The Gecode programming model for the complete semantics based on BMA.

⟨*BoolExpr*⟩	::=	⟨*x*⟩	Boolean variable
	|	! ⟨*BoolExpr*⟩	negation
	|	⟨*BoolExpr*⟩ && ⟨*BoolExpr*⟩	conjunction
	|	⟨*BoolExpr*⟩ || ⟨*BoolExpr*⟩	disjunction
	|	⟨*BoolExpr*⟩ == ⟨*BoolExpr*⟩	equivalence
	|	⟨*BoolExpr*⟩ != ⟨*BoolExpr*⟩	non-equivalence
	|	⟨*BoolExpr*⟩ >> ⟨*BoolExpr*⟩	implication
	|	⟨*BoolExpr*⟩ << ⟨*BoolExpr*⟩	reverse implication
	|	⟨*BoolExpr*⟩ ^ ⟨*BoolExpr*⟩	exclusive or

Fig. 2. The syntax for constructing Boolean expressions in *MiniModel*

Gecode using the built-in function **clause**, called in function **add_OR_Clause**.[2] It can be seen from these two programming models that Gecode is convenient and suitable for modeling argumentation problems. Using MiniModel and Int packages, we can easily program the Boolean constraint models of other semantics.

$$\begin{aligned}\mathcal{H}_{[\mathtt{CO1}]}(x_i) = & \bigwedge_{x_j \in \mathcal{R}^{+}(x_i)} (\mathbf{o}_i \vee \neg \mathbf{x}_j),\ \Big(\neg \mathbf{o}_i \vee \bigvee_{x_j \in \mathcal{R}^{+}(x_i)} \mathbf{x}_j\Big),\ \big(\neg \mathbf{x}_i \vee \neg \mathbf{o}_i\big), \\ & \bigwedge_{x_j \in \mathcal{R}^{+}(x_i)} (\neg \mathbf{x}_i \vee \mathbf{o}_j),\ \Big(\mathbf{x}_i \vee \bigvee_{x_j \in \mathcal{R}^{+}(x_i)} \neg \mathbf{o}_j\Big).\end{aligned} \quad (1)$$

[2] Absolutely, the package MiniMode also can handle CNF formulas, but in the current version of `argmat-mpg`, we merely use the package Int to encode the CNF formulas. This approach mainly refers from an example of Gecode, which implements a simple CNF SAT solver (see `http://www.gecode.org/doc/5.0.0/reference/sat_8cpp.html`).

```
// The constraint model for the CNF encoding of [CO1] with 2n Boolean variables
void encode_CO_cnf_2() {
    // Step 1: create 2 * argNum SAT varibles
    new_vars(2 * m_daf.getNumberOfArguments());

    // Step 2: get attacker list of each argument
    VecVecType attackerVec = m_daf.getAttackersVector();

    // Step 3: Post CNF constraints
    AMClause cls_binary;  // add binary clauses
    AMClause cls_admOpp, cls_rplusOpp;
    for (auto i = m_daf.getNumberOfArguments(); i--; ) {
        cls_binary.clear().addNegLit(i, _O(i));    // post ¬x_i ∨ ¬o_i
        add_OR_Clause(cls_binary);

        cls_admOpp.clear().addPosLit(i);          // add x_i to cls_admOpp
        cls_rplusOpp.clear().addNegLit(_O(i));    // add ¬o_i to cls_rplusOpp
        for (size_type j : attackerVec[i]) {
            cls_binary.clear().addPosNegLit(_O(j), i);  // post ¬x_i ∨ o_j
            add_OR_Clause(cls_binary);

            cls_binary.clear().addPosNegLit(_O(i), j);  // post o_i ∨ ¬x_j
            add_OR_Clause(cls_binary);

            cls_admOpp.addNegLit(_O(j));  // add ¬o_j to cls_admOpp
            cls_rplusOpp.addPosLit(j);    // add x_j to cls_rplusOpp
        }
        add_OR_Clause(cls_admOpp);    // post x_i ∨ ¬o_j1 ∨ ¬o_j2 ···
        add_OR_Clause(cls_rplusOpp);  // post ¬o_i ∨ x_j1 ∨ x_j2 ···
    }
}
```

Fig. 3. The Gecode programming model for the CO semantics using the CNF encoding of [CO1].

2 Branching

A branching in Gecode models defines the shape of the search tree. Gecode offers predefined variable-value branching: when calling branch() to post a branching, the third argument defines which variable is selected for branching, whereas the fourth argument defines which values are selected for branching. This technique is quite useful, since it can optimize a search tree so as to reduce the search time. In current version of argmat-mpg, we implement a novel variable selection strategy, which uses the topological information of argumentation frameworks to specify the variable order. argmat-mpg also allows to use the default and random variable order.

We also set up various value selection strategy for different semantics. For instance, the value selections for the maximal semantics (e.g., PR and ID), as well as the maximal range semantics (e.g., SST and STG), begin with Boolean value 1, since the extensions of these semantics are required to be able to accommodate as many arguments as possible, i.e., their corresponding Boolean vectors can contain as many 1s as possible. Similarly, we can solve GR semantics starting from Boolean value 0.

3 Search solutions

When the programming models and the branching methods of the semantics are specified, the next is to search the solutions of these models. Gecode provides a family of search engines with state-of-the-art performance to solve these models. Gecode also allows to develop new search engines to achieve the actual computing tasks. For solving stable and complete semantics, we use the built-in Depth-First Search (DFS) engine. For solving the maximal and the maximal range semantics, we implement two new search engines, respectively, both based on the built-in Branch-And-Bound (BAB) engine, which can be used to find a maximal (w.r.t. $\subseteq$) solution or range.

4 Supported options

`argmat-mpg` is implemented by C++ language, and supports all computational tasks of ICCMA-2017. It is multi-threading and cross-platform (Windows and Unix OS). Beside meeting the standard command interface of ICCMA-2017, `argmat-mpg` also provides some useful options. The explanations of these options are shown in Table 1, in which the options "-b" and "-e" allow to choose a branching strategy and a programming model. In current version, `argmat-mpg` supports the CNF-based programming models for all semantics, but merely supports the BMA-based programming models for ST and CO semantics. Future version may consider to support the BMA-based programming models for all semantics. The source codes submitted to ICCMA 2017 can be found on the website of our project `argumatrix`[3].

Table 1. The explanations for some useful options.

option	**value & explanation**
-b	specify the variables selection strategy for branching (default: **topo**) • **none**: using the default given variable selection strategy • **topo**: using topological variable selection strategy • **rand**: using random variable selection strategy (the random seed can be specified by "-r")
-e	specify the encoding options for the problem (default: **auto**) • **cnf**: using CNF encoding, supporting for all semantics • **auto**: auto select encoding, supporting for all semantics • **st-cnf**: using stable CNF encoding with n variables, supporting ST semantics • **st-ba**: using stable BMA encoding with n variables, supporting ST semantics • **co-cnf2**: using complete CNF encoding with $2n$ variables, supporting CO, PR, ID, and GR semantics • **co-ba**: using complete BMA encoding with n variables, supporting CO, PR, ID, and GR semantics • **ad-cnf**: using admissible CNF encoding with n variables, supporting PR and ID semantics • **ad-cnf2**: using admissible CNF encoding with $2n$ variables, supporting PR and ID semantics • **sst-cnf2**: using semistable CNF encoding with $2n$ variables, supporting SST semantics • **sst-cnf3**: using semistable CNF encoding with $3n$ variables, supporting SST semantics • **stg-cnf2**: using stage CNF encoding with $2n$ variables, supporting STG semantics
-o	specify where to output {**stdout**, **stderr**, **file**} (default: **stdout**)
-thrd	specify number of threads (default: **1**)
-r	specify the random seed, used for random variable selection (default: **0**)
-s	whether to show statistics, merely supports enumerating task {**false**, **0**, **true**, **1**} (default: **false**)

References

1. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Journal of Artificial Intelligence **77**(2) (September 1995) 321–357
2. Fuan, P., Guiming, L., JIANG, Z.: Encoding argumentation semantics by boolean algebra. IEICE Transactions on Information and Systems **100**(4) (2017) 838–848

[3] `https://sites.google.com/site/argumatrix/`