TOWARDS AN INCREMENTAL REENTRANT ASP-BASED SOLVER FOR ARGUMENTATION FRAMEWORKS

KAMILLO HUGH FERRY

Abstract. We devise a incremental ASP-based solving system for Dung's abstract argumentation frameworks. This means that we allow for dynamic changes to an input AF such as adding or removing arguments and attacks while retaining the state of the underlying grounder and solver.

This paper documents the design process and current state of the incremental solver system. Also, we details some of the problems that were encountered and open issues of the cosntructed system.

1. Introduction

The need for incremental algorithms arises when dealing with dynamically changing inputs where it is desirable to process each such change relatively quickly and where it is not desirable to process the input every time from scratch. One example for such a setting are knowledge bases, and in particular, abstract argumentation frameworks (AFs) that model a knowledge base.

Here, given an AF, we might have computed a set of possible extensions satisfying a certain semantic. But when we edit this AF, e. g. to react to a change in our knowledge base, and ask for the extensions of the resulting framework, we can try to keep information from previous solve iterations so we do not have to construct all extensions from scratch.

This approach to algorithmic problems trying to retain information between the runs of an algorithm and incrementally solving a dynamically changing instance has been studied in literature before [1, 2, 3].

An example taken from Patnaik and Immerman [3] (Example 3.2), when asking for the parity of a string of n bits, if we retain the parity bit between calculations, we can actually express the answer using propositional formulas when the allowed edits are flipping bits.

That is, suppose we have a data structure S representing the state of a string of n bits called S_x, and the parity bit b of this string. We

$$\mathsf{ins}(S,a): \quad \begin{aligned} S'_x &:= S_x \vee x = a \\ b' &:= (b \wedge S_a) \vee (\neg b \wedge \neg S_a) \end{aligned}$$

$$\mathsf{del}(S,a): \quad \begin{aligned} S'_x &:= S_x \wedge x \neq a \\ b' &:= (b \wedge \neg S_a) \vee (\neg b \wedge S_a) \end{aligned}$$

FIGURE (1). Operations ins and del for calculation of parity bits.

can define two operations $\mathsf{ins}(S,i)$ and $\mathsf{del}(S,i)$ that represent 'setting the i-th bit to 0 resp. 1' that edit S accordingly to the changes that were requested and those operations can be expressed by propositional formulae as in fig. 1.

The main idea behind this example is that given a specific problem, we might be able to identify how a change to the input affects the solution and what kind of intermediate information has to be kept during each computation.

Applying this to abstract argumentation frameworks, we build upon the ASP-based argumentation system ASPARTIX [**aspartix**] which already provides encodings for most common semantics of argumentation frameworks, albeit static ones.

ASPARTIX itself is implemented in the ASP system *clingo* [**clingo**], whose API for partial grounding and externally provided atoms is also heavily used to enable editing the input AF.

This is done by breaking apart the given encodings into parts that are affected by the insertion of an argument and parts that are affected by the insertion of an attack. Whenever an argument or attack is to be added, the respective partial encoding is applied.

The detailed process of modifying the encodings and modifying the state of arguments and attacks is described in section 3.

Examples for the usage of the current system are given in section 2. Current issues and possibilities for extension are described in sections 4 and 5

2. USAGE

Currently, our incremental system cannot be called from command-line. Instead, one has to instanciate a `IncrAFSolver` object in Python, as follows

```
from incraf import IncrAFSolver

af = IncrAFSolver(semantic, filename)
```

To create an `IncrAFSolver` object, one only needs to provide a `semantic` since an AF can also be built up succesively.

Arguments and attacks can be added by the respective `add` and `del` methods. Arguments can be named by strings or integers

The following example adds to an framework `af` two arguments named `2` and `b` and an attack between them while removing an attack from arguments `1` to `a`.

```
af.add_argument(2)
af.add_argument("b")
af.add_attack(2,"b")
af.del_attack(1,"a")
```

We can either enumerate all extensions satisfying the specified semantic with `solve_enum`, or test for credulous or skeptical satisfiability with `solve_cred` resp. `solve_skept`. To get the accepted arguments after calling one of the latter functions, one can use `extract_witness`.

The following snippet describes the process of testing for skeptical satisfiability and getting the skeptically accepted arguments.

```
af.solve_skept()

args = af.extract_witness()
```

More complete examples are given in the files `example_af1.jpynb` and `example_af2.jpynb`

3. Modification of the Aspartix encodings

To allow us to accept changes to our input, we need to make two adjustments to the way we feed the encodings to *clingo*.

- We need to add all atoms related to the input as `#external` atoms.
- The encoding for the semantic needs to be split in two parts `#program add_argument(v)` and `#program add_attack(v,u)`.

Declaring atoms as external instructs *clingo* to omit *certain simplifications* [**clingo_guide**]. For example, external atoms are not removed from rules even if they do not appear in the head of any rule.

Additionally, *clingo* allows us to provide the truth value for any external atom. We use this to switch off arguments or attacks that have been deleted.

The need to break apart an encoding nito parts related to arguments and attacks comes from the fact that the initial grounding only introduces rules for arguments and attacks that existed at that point. Even if we add new atoms, the rules related to them that tie them to a possible solution do not exist.

We cannot re-ground simply since then we also try to redefine rules that already existed. Yet the process of grounding can be (overly simplified) thought of plugging in the actual values for variables occuring in rules. Thus, breaking apart the encoding allows us to only introduce only the necessary rules whenever adding an attack or argument.

For this, *clingo* provides the `#program` statement to denote program blocks accepting parameters.

Take for example following encoding for naive semantics taken from ASPARTIX.

```
%% Guess a set S \subseteq A
in(X) :- not out(X), arg(X).
out(X) :- not in(X), arg(X).

%% S has to be conflict-free
:- in(X), in(Y), att(X,Y).

%% Check Maximality
okOut(X) :- in(Y), att(Y,X).
okOut(X) :- in(Y), att(X,Y).
okOut(X) :- att(X,X).
:- out(X), not okOut(X).
```

Obviously, the rules to guess a potential solution subset S are tied to a specific argument, while the rules checking for conflict-freeness are each tied to a specific attack. Also the rules checking for maximality depend on attacks.

This means that the guessing rules are added to the argument block while the checking rules are added to the attack block.

Special care needs to be taken for the last rule. Since this one only references an argument, we need to actually add this rule to the argument block. But since the `okOut(X)` atom does not occur in any rule head until there exists an attack involving `X`, we need to mark this atom as external.

The resulting incremental naive encoding then is as follows.

```
#program add_argument(v).
%% Guess a set S \subseteq A
in(v) :- not out(v),arg(v).
out(v) :- not in(v), arg(v).
#external okOut(v).
:- arg(v), out(v), not okOut(v).

#program add_attack(v, u).
%% S has to be conflict-free
:- in(v), in(u), att(v,u).

%% Check Maximality
okOut(u) :- in(v), att(v,u).
okOut(v) :- in(u), att(v,u).
okOut(v) :- att(v,v).
```

With an incremental encoding as the one above, we can now add an argument or an attack by first adding an external atom, if it did not exist before, and then performing the partial grounding of either `#program add_argument(v)` or `#program add_attack(v,u)`.

4. Issues

Currently, adding new attacks might not always work since *clingo* throws an error that an atom gets redefined. This is related to the initial problem of redefining rules, but this phenomenon only occurs after calling a `solve` method once.

Thus, the reason might be instead a conflict between a truth value that is already assigned to an atom and new rules causing this atom to change its assignment.

5. Open Problems

In additon to the blocking behaviour of *clingo* described in section 4 this solver does not provide a command-line interface yet.

Also, instead of trying to use *clingo* with an incremental encoding, it is also possible to instead use a static encoding that produces a solution and possibly additional intermediate information and retain that. Then, when processing changes to an instance, we just modify the information that was retained and only call the solver from scratch when necessary. This approach would be similar to the one presented by Mans and Mathieson [1] for incremental computations.

References

[1] Bernard Mans and Luke Mathieson. "Incremental Problems in the Parameterized Complexity Setting". In: *Theory of Computing Systems* 60.1 (), pp. 3–19. ISSN: 1432-4350, 1433-0490. DOI: 10.1007/s00224-016-9729-6.

[2] Peter Bro Miltersen et al. "Complexity models for incremental computation". In: *Theoretical Computer Science* 130.1 (), pp. 203–236. ISSN: 03043975. DOI: 10.1016/0304-3975(94)90159-7.

[3] Sushant Patnaik and Neil Immerman. "Dyn-FO: A Parallel, Dynamic Complexity Class". In: *Journal of Computer and System Sciences* 55.2 (), pp. 199–209. ISSN: 00220000. DOI: 10.1006/jcss.1997.1520.