TPLP **25** (2): 197–224, 2025. © The Author(s), 2025. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.
doi:10.1017/S1471068424000462 First published online 6 February 2025

On the Generalization of Learned Constraints for ASP Solving in Temporal Domains

JAVIER ROMERO, TORSTEN SCHAUB and KLAUS STRAUCH
University of Potsdam, Potsdam, Brandenburg, Germany,
(*e-mails:* javier@cs.uni-potsdam.de, torsten@cs.uni-potsdam.de, kstrauch@uni-potsdam.de)

submitted 29 April 2023; revised 15 October 2024; accepted 30 October 2024

Abstract

The representation of a temporal problem in answer set programming (ASP) usually boils down to using copies of variables and constraints, one for each time stamp, no matter whether it is directly encoded or expressed via an action or temporal language. The multiplication of variables and constraints is commonly done during grounding, and the solver is completely ignorant about the temporal relationship among the different instances. On the other hand, a key factor in the performance of today's ASP solvers is conflict-driven constraint learning. Our question in this paper is whether a constraint learned for particular time steps can be generalized and reused at other time steps, and ultimately whether this enhances the overall solver performance on temporal problems. Knowing well the domain of time, we study conditions under which learned dynamic constraints can be generalized. Notably, we identify a property of temporal representations that enables the generalization of learned constraints across all time steps. It turns out that most ASP planning encodings either satisfy this property or can be easily adapted to do so. In addition, we propose a general translation that transforms programs to fulfill this property. Finally, we empirically evaluate the impact of adding the generalized constraints to an ASP solver.

Keywords: answer set programming, answer set solving, temporal reasoning

1 Introduction

Although answer set programming (ASP; (Gelfond and Lifschitz 1988)) experiences increasing popularity in academia and industry, a closer look reveals that this concerns mostly static domains. There is still quite a chasm between ASP's level of development for addressing static and dynamic domains. This is because its modeling language as well as its solving machinery aim so far primarily at static knowledge, while dynamic knowledge is mostly dealt with indirectly via reductions to the static case. This also applies to dedicated dynamic formalisms like action and temporal languages (Gelfond and Lifschitz 1998; Aguado *et al*. 2013). In fact, their reduction to ASP or boolean satisfiability usually relies on translations that introduce a copy of each variable for each time step. The actual dynamics of the problem is thus compiled out and a solver treats the result as any other static problem.

We address this by proposing a way to (partly) break the opaqueness of the actual dynamic problem and equip an ASP solver with means for exploiting its temporal nature. More precisely, we introduce a method to strengthen the conflict-driven constraint learning (CDCL) framework of ASP solvers so that dynamic constraints learned for specific time points can be generalized to other points in time. These additional constraints can in principle reduce the search space and improve the performance of the ASP solvers.

We start in Section 2 showing our approach through an example. In Section 3, we review some background material. In Section 4, we introduce a simple but general language to reason about time in ASP. We define temporal problems, and characterize their solutions in terms of completion and loop nogoods, paralleling the approach to regular ASP solving (Gebser *et al.* 2012). In Section 5, using this language, we study conditions under which learned constraints can be generalized to other time steps. In Section 6, we identify a sufficient condition for the generalization of all learned constraints. In Section 7, we present a translation from temporal programs that generates programs satisfying that condition. In Section 8, we empirically evaluate the impact of adding the generalized constraints to the ASP solver *clingo*. We conclude in Section 9.

Our work can be seen as a continuation of the approach of *ginkgo* (Gebser *et al.* 2016), which also aimed at generalizing temporal constraints but resorted to an external inductive proof method (in ASP) for warranting correctness. More generally, a lot of work has been conducted over recent years on lazy ASP solving (Lefèvre *et al.* 2017; Palù *et al.* 2009; Weinzierl *et al.* 2020). Notably, conflict generalization was studied from a general perspective in (Comploi-Taupe *et al.* 2020), dealing with several variables over heterogeneous domains. Lazy grounding via propagators was investigated in (Cuteri *et al.* 2020; Mazzotta *et al.* 2022; Dodaro *et al.* 2023). Finally, it is worth mentioning that the usage of automata, as done in (Cabalar *et al.* 2021), completely abolishes the use of time points. A detailed formal and empirical comparative study of these approaches is interesting for future work.

This is an extended version of the conference paper (Romero *et al.* 2022) presented at RuleML+RR 2022. The main new contribution is the identification in Section 6 of a property of temporal problems that enables the generalization of learned constraints across all time points without the need of any translation. This significantly improves the applicability of our approach. In fact, this property is satisfied by the planning domains that we considered in our empirical evaluation of (Romero *et al.* 2022). Given this, we ran those experiments again, but this time using the original encodings, only slightly modified to satisfy the mentioned property. In Section 7 we also introduce a new translation to programs that satisfy that property. This translation is both more general and easier to understand than the one presented in our previous conference paper (Romero *et al.* 2022), which can be found in the supplementary material along with the proofs of the theoretical results.

2 An example

Our running example in this section is the *Blocks World* problem, represented in the STRIPS subset of the planning domain definition language (PDDL) planning language (McDermott 1998). We follow the approach of the *plasp* system (Dimopoulos *et al.* 2017), which translates a PDDL description into a set of facts, combines these facts with a meta-encoding implementing the PDDL semantics, and solves the resulting program using the ASP solver *clingo*.

Our instance consists of three blocks, `a`, `b`, and `c`. Initially, `a` is on top of `b`, that is on top of `c`, that is on the table. The goal is to rearrange the blocks in reverse order. This is represented by the following set of facts:

```
 block(a;b;c). init(clear(a)). init( handempty).
init(on(a,b)). init( on(b,c)). init(ontable(c)).
goal(on(b,a)). goal( on(c,b)). goal(ontable(a)).
```

There are four actions in the domain: `pick_up(X)` picks up a block `X` that is on the table, `put_down(X)` puts down a block `X` on the table, `stack(X,Y)` stacks the block `X` on top of block `Y`, and `unstack(X,Y)` undoes that operation. The following lines specify the action `stack(X,Y)` in terms of its `pre`conditions, `del`ete (or negative) effects and `add`itive (or positive) effects – where '`B`' is a shorthand for the body '`block(X), block(Y)`':

```
       action(stack(X,Y)) :- B. pre(stack(X,Y),holding(X)) :- B.
add(stack(X,Y), clear(X)) :- B. pre(stack(X,Y),   clear(Y)) :- B.
add(stack(X,Y),handempty) :- B. del(stack(X,Y),holding(X)) :- B.
add(stack(X,Y),   on(X,Y)) :- B. del(stack(X,Y),   clear(Y)) :- B.
```

The specification of the rest of the actions is available in the supplementary material. The following meta-encoding gives meaning to the previous rules:

```
1  holds(F,0) :- init(F).
2  { occ(A,T) : action(A) } = 1 :- T=1..n.
3  :- occ(A,T), pre(A,F), not holds(F,T-1).
4  holds(F,T) :- occ(A,T), add(A,F).
5  holds(F,T) :- holds(F,T-1), T=1..n,
6                    not occ(A,T) : del(A,F).
7  :- goal(F), not holds(F,n).
```

Line 1 specifies which fluents `F` hold at time step `0`, Line 2 generates one action `A` per time step `T` between `1` and some constant `n`, Line 3 forbids the occurrence of an action `A` if some of its preconditions do not hold, Line 4 defines the positive effects of an action `A`, Lines 5-6 define inertia for all fluents `F`, and Line 7 enforces that every goal fluent `F` holds at the last time point `n`.

The shortest plan for this problem requires six actions, one to pick-up or unstack every block, and another to put it down or stack it. Accordingly, when the constant `n` has the value `6`, our program has a unique stable model that includes the atoms `occ(unstack(a,b),1)`, `occ(put_down(a),2)`, `occ(unstack(b,c),3)`, `occ(stack(b,a),4)`, `occ(pick_up(c),5)`, and `occ(stack(c,b),6)`.

Such a stable model can be computed by the ASP solver *clingo*, that is based in the *ground-and-solve* approach to ASP solving. In the first step, *clingo* grounds the logic program by replacing the rules with variables with their ground (variable-free) instantiations. The goal of this step is to generate a ground program that is as compact as possible, while preserving the stable models of the original program. As an example, consider the precondition rule in Line 3 of the meta-encoding for action `stack(b,a)`. At time point `2`, *clingo* generates two ground instances, one for each precondition of the action:

```
:- occ(stack(b,a),2), not holds(  clear(a),1).
:- occ(stack(b,a),2), not holds(holding(b),1).
```

At time point `1`, however, the corresponding first instance is not generated, as *clingo* infers that `holds(clear(a),0)` must be true due to the fact `init(clear(a))` and the rule in Line 1. In turn, the corresponding second instance is simplified to

```
:- occ(stack(b,a),1).
```

since *clingo* deduces that `holds(holding(b),0)` cannot be true, as no rule in the program can derive it.

In the second step, *clingo* uses a Conflict-Driven Nogood Learning (CDNL) algorithm to search for stable models of the ground program generated before. Whenever the algorithm backtracks, it learns a new constraint, also called a *nogood*, that is satisfied by all stable models. These learned constraints are crucial for the performance of the solver, as they prevent it from making the same mistakes more than once. We explain this in more detail in Section 3. As an example, while solving our program, *clingo* could learn the constraint

```
:- holds(on(a,b),2), holds(on(b,c),2), not holds(on(b,c),4).      (1)
```

representing that if at time point `2` block `a` is on top of block `b`, and `b` is on top of block `c`, then at time point `4` block `b` must remain on top of `c`. Note that although the constraint is specific to time points `2` and `4`, the solver learned it using rules that are replicated at every time point. Then, the main question of this paper is:

Can we generalize such learned constraints to other time points?

And if so:

What impact does this have on the performance of ASP solvers?

We can generalize (1) to all time steps in the interval `0..n` with the following constraint:

```
:- holds(on(a,b),T-2), holds(on(b,c),T-2),                      (2)
   not holds(on(b,c),T), T=2..n.
```

This constraint can be safely added to our program, since it does not eliminate any of its stable models. Ideally, the solver could detect this automatically and learn this generalized constraint directly instead of (1). This could help it to prune the search space and find solutions faster. However, such generalization is not always easy, for two main reasons.

The first reason is that, as we have seen, *clingo* does not replicate *exactly* the same ground rules at every time point. Since the constraints learned by the solver depend on these ground rules, a constraint learned at one time step may not apply to others. For example, the solver can learn

```
:- not holds(on(b,c),2).
```
(3)

which corresponds to constraint (1) for step `2`, without the atoms `holds(on(a,b),0)` and `holds(on(b,c),0)`. During grounding, *clingo* infers that these atoms are true and simplifies the ground rules where they appear accordingly. As a result, the constraints learned from these simplified rules are also simpler – compare (3) with (1). Since these simplifications are not correct for all time points, the generalization of (3) to all time points

```
:- not holds(on(b,c),T), T=0..n.
```

is also not correct. In fact, it would eliminate all stable models.

We address this issue manually, modifying the meta-encoding in such a way that the ground rules are truly the same for all time steps. We achieve this by replacing the rule in Line 1 by the choice rules

```
{ holds(F,0) } :-  init(F).
{ holds(F,0) } :- add(A,F).
```

that generate all possible initial states, and by eliminating the integrity constraint of Line 8. Once this is done, the ground instantiations at all time steps become the same. For example, the ground instantiation of the rule in Line 3 for action `stack(b,a)` at time point `1` becomes

```
:- occ(stack(b,a),1), not holds(  clear(a),0).
:- occ(stack(b,a),1), not holds(holding(b),0).
```

and instead of learning (3), *clingo* would learn the constraint

```
:- holds(on(a,b),0), holds(on(b,c),0), not holds(on(b,c),2).
```
(4)

that can be safely generalized to (2). Observe that the modified program no longer determines the values of the fluents at the initial and final situations. Instead, these values are specified by passing to *clingo* an additional set of so-called *assumptions*. Notably, these assumptions preserve the stable models of the original program but do not interfere with the grounding process. Hence, the ground rules remain the same across all time steps.

We formalize this approach in Section 4 using *temporal logic problems*, which consist of *temporal logic programs* defining the ground rules that are replicated at each time step, and partial assignments representing the initial and final states. Additionally, we study the *temporal nogoods* associated with temporal logic programs. Solving with assumptions is introduced in Section 3, while its application to temporal logic problems is described in Section 5.

To illustrate the second issue, let us extend our example by introducing a counter that is incremented with each action. We modify Line 4 of the meta-encoding to represent conditional effects of actions as follows:

```
holds(F,T) :- occ(A,T), add(A,F), holds(G,T-1) : cond(A,F,G).
```

The atom `cond(A,F,G)` expresses that for action `A` to produce effect `F`, the fluent `G` must have been true in the previous state. The counter can be represented as a conditional effect of all actions:

```
                 add(A,counter(1)) :- action(A).
cond(A,counter(T),counter(T-1)) :- action(A), T = 2..m.
```

Actions make `counter(1)` true, and for `T` between `2` and `m`, they also make `counter(T)` true if `counter(T-1)` was true in the previous state. Given this extension, *clingo* can learn the constraint

```
:- not holds(counter(2),2).                                    (5)
```

as `counter(2)` always holds at time point `2` in all stable models. However, the generalization to all time points

```
:- not holds(counter(2),T), T=0..n.                            (6)
```

would be incorrect, as `counter(2)` does not have to hold at time points `0` or `1`. This example shows that, even when all rules are replicated across all time steps, we cannot always generalize learned constraints to all time steps. This raises the question: when can we generalize learned constraints to other – or all – time steps?

We provide a first answer in Section 5, after formalizing the problem of generalizing learned constraints. Assume that the time steps of the rules used to learn a constraint are known. This information allows us to to determine the time steps to which a learned constraint can we generalized. For instance, if constraint (5) — about fluent `counter(2)` at step `2` — was learned using rules from time steps `1` and `2`, we could generalize it to the interval `T = 2..n`, because the rules used at `1` and `2` have corresponding copies at `T-1` and `T`:

```
:- not holds(counter(2),T), T=2..n.                            (7)
```

However, we could not generalize the constraint to `T = 1`, as no copies of the rules exist for time step `0`. As another example, if (5) was learned using rules of steps `2` and `3`, then the generalized constraint would apply to the interval `T = 1..n-1`, as the rules at `2` and `3` would have their corresponding copies at `T` and `T + 1`, and there are no copies at `n + 1`. Clearly, a drawback of this approach is that it requires tracking the time steps of the rules used to learn each constraint. Consequently, its implementation would require the modification of the inner machinery of *clingo*.

To avoid this, in Section 6 we investigate under which conditions can we generalize learned constraints to all time steps. In our previous examples, we could not generalize (5) to (6) because there were no copies of the rules at step `0` or step `n + 1`. But what would happen if the program behaved *as if* those copies existed? It turns out that in such a case we could generalize the learned nogoods to all time steps. How can we capture these cases? We define for each temporal program a *transition graph* whose edges correspond to the transitions between states. If every initial state has a predecessor in that graph, it is *as if* the rule copies applied also to those initial states. Likewise, if every state reachable from an initial state has a successor, it is *as if* the rule copies applied to those final states as well. In Section 6 we define temporal programs as *internal* if their transition graph satisfies these properties, and we show that for internal programs all learned nogoods can be generalized to *all* time steps.

In practice, most planning representations that we have encountered are either internal or can be easily adapted to become internal. For instance, to adapt our meta-encoding, it is sufficient to modify Line 2 by replacing the comparison operator = by <=, allowing the non-execution of actions at any time step. The inertia rule in Lines 5-6 ensures that fluents persist whenever no action occurs. Thus, initial states, where no action occurs, have themselves as predecessors. Similarly, reachable states have a successor state where no action occurs and all fluents persist. This small adjustment makes the representation internal. Once it is applied, constraint (5) can no longer be learned, as it does not hold in all stable models, that is there are stable models where no action occurs at time steps `1` or `2`, and `counter(2)` does not hold at `2`. Instead, the solver can learn the following constraint:

```
:- not holds(counter(2),2), occ(unstack(a,b),1),
   occ(putdown(a),2).
```

which can be safely generalized to

```
:- not holds(counter(2),T), occ(unstack(a,b),T-1),
   occ(putdown(a),T), T=1..n.
```

We consider this constraint, with `T=1..n`, a generalization across all time steps because its ground instances include atoms over all steps. In this case, extending the generalization to additional steps would be pointless. For other constraints, it could even be incorrect due to the semantics of negation.

To handle cases where a temporal representation is not internal, in Section 7 we introduce a translation that transforms any temporal program into an internal one. In our example, the translation adds the following rules:

```
{  lambda(T) } :- T=1..n.
{ holds(F,T) } :-  init(F), T=1..n, not lambda(T).
{ holds(F,T) } :-  add(A,F), T=1..n, not lambda(T).
{   occ(A,T) } :- action(A), T=1..n, not lambda(T).
```

Additionally, the translation inserts the atom `lambda(T)` into the body of each rule in the meta-encoding, and assumes that `lambda` is false at time step `0`. This translation ensures that initial states – where `lambda` is false – have themselves as predecessors, and reachable states have themselves – after erasing `lambda --` as successors. Once this translation is applied, *clingo*, instead of learning constraint (5), could learn

```
:- not holds(counter(2),2), lambda(2), lambda(1).
```

that could be generalized to

```
:- not holds(counter(2),T), lambda(T), lambda(T-1), T=1..n.      (8)
```

Furthermore, the constraints learned using the translation can be applied to the original program, after simplifying away the atoms over `lambda` and adapting the interval of `T` to `T`$=$`2..n`. In our example, (8) can be simplified to (7), that can be safely added to our original program.

3 Background

We review the material from (Gebser *et al.* 2007) about solving normal logic programs and adapt it for our purposes to cover normal logic programs with choice rules and integrity constraints over some set $\mathcal{A}$ of atoms.

A *rule* r has the form $H \leftarrow B$ where B is a set of literals over $\mathcal{A}$, and H is either an atom $a \in \mathcal{A}$, and we call r a *normal rule*, or $\{a\}$ for some atom $a \in \mathcal{A}$, making r a *choice rule*, or $\bot$, so that r is an *integrity constraint*. We usually drop braces from rule bodies B and drop the arrow $\leftarrow$ when B is empty. We use the extended choice rule $\{a_1; \dots ; a_n\} \leftarrow B$ as a shorthand for the choice rules $\{a_1\} \leftarrow B$, ..., $\{a_n\} \leftarrow B$, and for some set of atoms $X \subseteq \mathcal{A}$ we denote by $\mathit{Choice}(X)$ the set of choice rules $\{\{a\} \leftarrow \mid a \in X\}$. A *program* Π is a set of rules. By Π^n, Π^c, and Π^i we denote its normal rules, choice rules and integrity constraints, respectively. Semantically, a logic program induces a collection of *stable models*, which are distinguished models of the program determined by the stable models semantics (see Gebser *et al.* 2012 and Gelfond and Lifschitz 1988 for details).

For a rule r of the form $H \leftarrow B$, let $h(r) = a$ be the *head* of r if H has the form a or $\{a\}$ for some atom $a \in \mathcal{A}$, and let $h(r) = \bot$ otherwise. Let $\mathit{Bd}(r) = B$ be the *body* of r, $\mathit{Bd}(r)^+ = \{a \mid a \in \mathcal{A}, a \in B\}$ be the *positive body* of r, and $\mathit{Bd}(r)^- = \{a \mid a \in \mathcal{A}, \neg a \in B\}$ be the *negative body* of r. The set of atoms occurring in a rule r and in a logic program Π are denoted by $\mathit{At}r$ and $\mathit{At}\Pi$, respectively. The set of bodies in Π is $\mathit{Bd}(\Pi) = \{\mathit{Bd}(r) \mid r \in \Pi\}$. For regrouping rule bodies sharing the same head a, we define $\mathit{Bd}(a) = \{\mathit{Bd}(r) \mid r \in \Pi, h(r) = a\}$, and by $\mathit{Bd}^n(a)$ we denote the restriction of that set to bodies of normal rules, that is $\{\mathit{Bd}(r) \mid r \in \Pi^n, h(r) = a\}$.

A Boolean *assignment* S over a set $\mathcal{D}$, called the domain of S, is a set $\{\sigma_1, \dots, \sigma_n\}$ of *signed literals* σ_i of the form $\mathbf{T}a$ or $\mathbf{F}a$ for some $a \in \mathcal{D}$ and $1 \leq i \leq n$; $\mathbf{T}a$ expresses that a is *true* and $\mathbf{F}a$ that it is *false*. We omit the attribute *signed* for literals whenever clear from the context. We denote the complement of a literal σ by $\overline{\sigma}$, that is, $\overline{\mathbf{T}a} = \mathbf{F}a$ and $\overline{\mathbf{F}a} = \mathbf{T}a$. Given this, we access true and false propositions in S via $S^{\mathbf{T}} = \{a \in \mathcal{D} \mid \mathbf{T}a \in S\}$ and $S^{\mathbf{F}} = \{a \in \mathcal{D} \mid \mathbf{F}a \in S\}$. We say that a set of atoms X is consistent with an assignment S if $S^{\mathbf{T}} \subseteq X$ and $S^{\mathbf{F}} \cap X = \emptyset$. In our setting, a *nogood* is a set $\{\sigma_1, \dots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment containing $\sigma_1, \dots, \sigma_n$. Accordingly, the nogood for a body B, denoted by $ng(B)$, is $\{\mathbf{T}a \mid a \in B^+\} \cup \{\mathbf{F}a \mid a \in B^-\}$. We say that an assignment S over $\mathcal{D}$ is *total* if $S^{\mathbf{T}} \cup S^{\mathbf{F}} = \mathcal{D}$ and $S^{\mathbf{T}} \cap S^{\mathbf{F}} = \emptyset$. A total assignment S over $\mathcal{D}$ is a *solution* for a set Δ of nogoods, if $\delta \not\subseteq S$ for all $\delta \in \Delta$. A set Δ of nogoods *entails* a nogood δ if $\delta \not\subseteq S$ for all solutions S over $\mathcal{D}$ for Δ, and it entails a set of nogoods ∇ if it entails every nogood $\delta \in \nabla$ in the set.

We say that a nogood δ is a *resolvent* of a set of nogoods Δ if there is a sequence of nogoods $\delta_1, \dots, \delta_n$ with $n \geq 1$ such that $\delta_n = \delta$, and for all i such that $1 \leq i \leq n$, either $\delta_i \in \Delta$, or there are some δ_j, δ_k with $1 \leq j < k < i$ such that $\delta_i = (\delta_j \setminus \{\sigma\}) \cup (\delta_k \setminus \{\overline{\sigma}\})$ for some signed literal σ. In this case, we say that the sequence $\delta_1, \dots, \delta_n$ is a proof of δ_n. We say that a signed literal σ is unit resulting for a nogood δ and an assignment S if $\delta \setminus S = \{\sigma\}$ and $\overline{\sigma} \notin S$. For a set of nogoods Δ and an assignment S, unit propagation is the process of extending S with unit-resulting literals until no further literal is unit resulting for any nogood in Δ.

Inferences in ASP can be expressed in terms of atoms and rule bodies. We begin with nogoods capturing inferences from the Clark completion. For a body $B = \{a_1, \dots, a_m, \neg a_{m+1}, \dots, \neg a_n\}$, we have that $\delta(B) = \{\mathbf{F}B, \mathbf{T}a_1, \dots, \mathbf{T}a_m, \mathbf{F}a_{m+1}, \dots, \mathbf{F}a_n\}$ and $\Delta(B) = \{\, \{\mathbf{T}B, \mathbf{F}a_1\}, \dots, \{\mathbf{T}B, \mathbf{F}a_m\}, \{\mathbf{T}B, \mathbf{T}a_{m+1}\}, \dots, \{\mathbf{T}B, \mathbf{T}a_n\} \,\}$. For an atom a such that $Bd^n(a) = \{B_1, \dots, B_k\}$, we have that $\Delta(a) = \{\, \{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\} \,\}$, and if $Bd(a) = \{B_1, \dots, B_k\}$ then $\delta(a) = \{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\}$. Given this, the *completion nogoods* of a logic program Π are defined as follows:

$$\begin{aligned}\Delta_\Pi = &\{\delta(B) \mid B \in Bd(\Pi \setminus \Pi^i)\} \cup \{\delta \in \Delta(B) \mid B \in Bd(\Pi \setminus \Pi^i)\}\\ &\cup \{\delta(a) \mid a \in At\Pi\} \cup \{\delta \in \Delta(a) \mid a \in At\Pi\}\\ &\cup \{ng(B) \mid B \in Bd(\Pi^i)\}\end{aligned}$$

Choice rules of the form $\{a\} \leftarrow B$ are considered by not adding the corresponding nogood $\{\mathbf{F}a, \mathbf{T}B\}$ to $\Delta(a)$, and integrity constraints from Π^i of the form $\bot \leftarrow B$ are considered by adding directly their corresponding nogood $ng(B)$. The definition of the *loop nogoods* Λ_Π, capturing the inferences from loop formulas, is the same as in (Gebser *et al.* 2007). We do not specify them here since they do not pose any special challenge to our approach, and they are not needed in our (tight) examples.

To simplify the presentation, we slightly deviate from (Gebser *et al.* 2007) and consider a version of the nogoods of a logic program where the occurrences of the empty body are simplified. Note that $\delta(\emptyset) = \{\mathbf{F}\emptyset\}$ and $\Delta(\emptyset) = \emptyset$. Hence, if $\emptyset \in Bd(\Pi)$ then any solution to the completion and loop nogoods of Π must contain $\mathbf{T}\emptyset$. Based on this, we can delete from $\Delta_\Pi \cup \Lambda_\Pi$ the nogoods that contain $\mathbf{F}\emptyset$ and eliminate the occurrences of $\mathbf{T}\emptyset$ from the others. Formally, we define the set of (simplified) nogoods for Π as:

$$\Sigma_\Pi = \{\delta \setminus \{\mathbf{T}\emptyset\} \mid \delta \in \Delta_\Pi \cup \Lambda_\Pi, \mathbf{F}\emptyset \notin \delta\}.$$

To accommodate this change, for a program Π, we fix the domain $\mathcal{D}$ of the assignments to the set $At\Pi \cup (Bd(\Pi) \setminus \emptyset)$. Given this, the stable models of a logic program Π can be characterized by the nogoods Σ_Π for that program. This is made precise by the following theorem, which is an adaptation of Theorem 3.4 from (Gebser *et al.* 2007) to our setting.

Theorem 1.
Let Π be a logic program. Then, $X \subseteq At\Pi$ is a stable model of Π iff $X = S^{\mathbf{T}} \cap At\Pi$ for a (unique) solution S for Σ_Π.

To compute the stable models of a logic program Π, we apply the algorithm *CDNL-ASP*(Π) from (Gebser *et al.* 2007) implemented in the ASP solver *clingo*. The algorithm searches for a solution S to the set of nogoods Σ_Π, and when it finds one it returns the corresponding set of atoms $S^{\mathbf{T}} \cap At\Pi$. *CDNL-ASP* maintains a current assignment S and a current set of learned nogoods ∇, both initially empty. The main loop of the algorithm starts by applying unit propagation to $\Sigma_\Pi \cup \nabla$, possibly extending S. Every derived literal is "implied" by some nogood $\delta \in \Sigma_\Pi \cup \nabla$, which is stored in association with the derived literal. This derivation may lead to the violation of another nogood. This situation is called *conflict*. If propagation finishes without conflict, then a (heuristically chosen) literal can be added to S, provided that S is partial, while otherwise S represents a solution and can be directly returned. On the other hand, if there is a conflict, there are two possibilities. Either it is a top-level conflict, independent of heuristically chosen literals, in which case the algorithm returns *unsatisfiable*. Or, if that is not the case, the conflict is analyzed to calculate a conflict nogood δ, that is added to ∇. More in detail, δ is a resolvent of the set of nogoods associated with the literals derived after the last heuristic choice. Hence, every learned nogood δ added to ∇ is a resolvent of $\Sigma_\Pi \cup \nabla$ and, by induction, it is also a resolvent of Σ_Π. After recording δ, the algorithm backjumps to the earliest stage where the complement of some formerly assigned literal is implied by δ, thus triggering propagation and starting the loop again.

This algorithm has been extended for solving under assumptions (Eén and Sörensson 2003). In this setting, the procedure $\mathit{CDNL\text{-}ASP}(\Pi, S)$ receives additionally as input a partial assignment S over $At\Pi$, the so-called assumptions, and returns some stable model of Π that is consistent with S. To accommodate this extension, the algorithm simply decides first on the literals from S, and returns *unsatisfiable* as soon as any of these literals is undone by backjumping. No more changes are needed. Notably, the learned nogoods are still resolvents of Δ_Π, that are independent of the set of assumptions S.

4 Temporal programs, problems and nogoods

We introduce a simple language of temporal logic programs to represent temporal problems. These programs represent the dynamics of a temporal domain by referring to two time steps: the current step and the previous step. We refer to the former by atoms from a given set $\mathcal{A}$, and to the latter by atoms from the set $\mathcal{A}' = \{a' \mid a \in \mathcal{A}\}$, that we assume to be disjoint from $\mathcal{A}$. We define a *state* as a subset of $\mathcal{A}$. Following the common-sense flow of time, normal or choice rules define the atoms of the current step in terms of the atoms of both the current and the previous step. Integrity constraints forbid some states, possibly depending on their previous state. Syntactically, a temporal logic program Π over $\mathcal{A}$ has the form of a (non-temporal) logic program over $\mathcal{A} \cup \mathcal{A}'$ such that for every rule $r \in \Pi$, if $r \in \Pi^n \cup \Pi^c$ then $h(r) \in \mathcal{A}$, and otherwise $(Bd(r)^+ \cup Bd(r)^-) \cap \mathcal{A} \neq \emptyset$. Given that temporal logic programs over $\mathcal{A}$ can also be seen as (non-temporal) logic programs over $\mathcal{A} \cup \mathcal{A}'$, in what follows we may apply the notation of the latter to the former.

This language is designed to capture the core of the translations to ASP of action and temporal languages. For instance, in our introductory PDDL example (McDermott 1998), a temporal logic program can represent the transition between `T-1` and `T` defined by the rules in Lines 2-6 of the meta-encoding. In essence, this temporal program corresponds to the ground instantiation of those rules at a single time point. Temporal programs are also closely related to the present-centered programs used in the implementation of the temporal ASP solver *telingo* (Cabalar *et al.* 2019), or to the programs that define the transitions in action languages (Lee *et al.* 2013).

Example 1.
Our running example is the temporal logic program Π_1 over $\mathcal{A}_1 = \{a, b, c, d\}$ that consists only of choice rules and integrity constraints:

$$\begin{array}{rl}
\{a; b; c; d\} \leftarrow & \qquad \bot \leftarrow a', \neg b \\
\bot \leftarrow \neg b', b & \qquad \bot \leftarrow \neg c', a \\
\bot \leftarrow d', b & \qquad \bot \leftarrow c, \neg d \\
\bot \leftarrow \neg a', \neg c & \qquad \bot \leftarrow \neg a', c', \neg a
\end{array}$$

Temporal logic programs Π can be instantiated to specific time intervals. We introduce some notation for that. Let m and n be integers such that $1 \le m \le n$, and $[m, n]$ denote the set of integers $\{i \mid m \le i \le n\}$. For $a \in \mathcal{A}$, the symbol $a[m]$ denotes the atom a_m, and for $a' \in \mathcal{A}'$, the symbol a'[m] denotes the atom a_{m-1}. For a set of atoms $X \subseteq \mathcal{A} \cup \mathcal{A}'$, $X[m]$ denotes the set of atoms $\{a[m] \mid a \in X\}$, and $X[m, n]$ denotes the set of atoms $\{a[i] \mid p \in X, i \in [m, n]\}$. For a rule r over $\mathcal{A} \cup \mathcal{A}'$, the symbol $r[m]$ denotes the rule that results from replacing in r every atom $a \in \mathcal{A} \cup \mathcal{A}'$ by $a[m]$, and $r[m, n]$ denotes the set of rules $\{r[i] \mid i \in [m, n]\}$. Finally, for a temporal program Π, $\Pi[m]$ is $\{r[m] \mid r \in \Pi\}$, and $\Pi[m, n]$ is $\{\Pi[i] \mid i \in [m, n]\}$.

Example 2.
The instantiation of Π_1 at 1, denoted by $\Pi_1[1]$, is

$$\begin{array}{rl}
\{a_1; b_1; c_1; d_1\} \leftarrow & \qquad \bot \leftarrow a_0, \neg b_1 \\
\bot \leftarrow \neg b_0, b_1 & \qquad \bot \leftarrow \neg c_0, a_1 \\
\bot \leftarrow d_0, b_1 & \qquad \bot \leftarrow c_1, \neg d_1 \\
\bot \leftarrow \neg a_0, \neg c_1 & \qquad \bot \leftarrow \neg a_0, c_0, \neg a_1
\end{array}$$

The programs $\Pi_1[i]$ for $i \in \{2, 3, 4\}$ are the same, except that the subindex 1 is replaced by i, and the subindex 0 is replaced by $i - 1$. The instantiation of Π_1 at $[1, 4]$, denoted by $\Pi_1[1, 4]$, is $\Pi_1[1] \cup \Pi_1[2] \cup \Pi_1[3] \cup \Pi_1[4]$.

To represent temporal reasoning problems, temporal programs are complemented by assignments I and F that partially or completely describe the initial and the final state of a problem. Formally, a *temporal logic problem* over some set of atoms $\mathcal{A}$ is a tuple (Π, I, F) where Π is a temporal logic program over $\mathcal{A}$, and I and F are assignments over $\mathcal{A}$. A solution to such a problem is a sequence of states that is consistent with the dynamics described by Π and with the information provided by I and F. The possible sequences of states of length n, for some integer $n \ge 1$, are represented by the *generator program* for Π and n, denoted by $gen(\Pi, n)$, that consists of the rules:

$$Choice(\mathcal{A})[0] \cup \Pi[1, n].$$

A *solution* to a temporal problem (Π, I, F) is defined as a pair (X, n), where n is an integer such that $n \ge 1$ and X is a stable model of $gen(\Pi, n)$ consistent with $I[0] \cup F[n]$.

Temporal problems can be used to formalize planning problems, using a temporal logic program Π of the form described above, a total assignment I that assigns a value to every possible atom (action occurrences are made false initially), and a partial assignment F to fix the goal. The solutions of the temporal problem correspond to the plans of the planning problem.

Example 3.
The temporal problem $(\Pi_1, \emptyset, \emptyset)$ has three solutions of length 4: $(X, 4)$, $(X \cup \{d_2\}, 4)$, and $(X \cup \{b_3\}, 4)$, where X is the set of atoms $\{a_0, b_0, c_0, a_1, b_1, b_2, c_3, d_3, a_4, c_4, d_4\}$.

To pave the way to the nogood characterization of temporal logic problems, we define the transition program $\mathit{trans}(\Pi)$ of a temporal logic program Π as the (non-temporal) logic program $\mathit{Choice}(A') \cup \Pi$ over $\mathcal{A} \cup \mathcal{A}'$. Each stable model of this program represents a possible transition between a previous and a current step, where the former is selected by the additional choice rules over atoms from $\mathcal{A}'$, and the latter is determined by the rules of Π, interpreted as non-temporal rules.

Example 4.
The transition program $\mathit{trans}(\Pi_1)$ is the (non-temporal) program $\Pi_1 \cup \{\{a'; b'; c'; d'\} \leftarrow\}$ over $\mathcal{A}_1 \cup \mathcal{A}_1{}'$. Some stable models of $\mathit{trans}(\Pi_1)$ are $\{a', b', c', a, b\}$ and $\{c', d', a, c, d\}$, that correspond to the transitions to step 1 and step 4 of the solution $(X, 4)$, respectively.

Next, we introduce temporal nogoods and their instantiation. Given a temporal logic program Π over $\mathcal{A}$, a temporal nogood over $\mathcal{A} \cup \mathit{Bd}(\Pi)$ has the form of a (non-temporal) nogood over $\mathcal{A} \cup \mathcal{A}' \cup \mathit{Bd}(\Pi)$. For a temporal nogood δ over $\mathcal{A} \cup \mathit{Bd}(\Pi)$ and an integer $n \geq 1$, the instantiation of δ at n, denoted by $\delta[n]$, is the nogood that results from replacing in δ any signed literal $\mathbf{T}\alpha$ ($\mathbf{F}\alpha$) by $\mathbf{T}\alpha[n]$ (by $\mathbf{F}\alpha[n]$, respectively). We extend this notation to sets of nogoods and to intervals like we did above. For example, $\delta_1 = \{\mathbf{F}b', \mathbf{T}b\}$ is a temporal nogood over $\mathcal{A}_1 \cup \mathit{Bd}(\Pi_1)$, and $\delta_1[1, 2]$ is $\{\{\mathbf{F}b_0, \mathbf{T}b_1\}, \{\mathbf{F}b_1, \mathbf{T}b_2\}\}$. By $\mathit{step}(\delta)$ we denote the steps of the literals occurring in δ, that is $\mathit{step}(\delta) = \{i \mid \mathbf{T}p_i \in \delta \text{ or } \mathbf{F}p_i \in \delta\}$. For example, $\mathit{step}(\{\mathbf{F}b_0, \mathbf{F}b_1, \mathbf{T}b_2\}) = \{0, 1, 2\}$.

We are now ready to define the temporal nogoods for a temporal logic program Π over $\mathcal{A}$. Recall that $\mathit{trans}(\Pi)$ is a (non-temporal) logic program over $\mathcal{A} \cup \mathcal{A}'$, whose corresponding nogoods are denoted by $\Sigma_{\mathit{trans}(\Pi)}$. Then, the set of *temporal nogoods* for Π, denoted by Ψ_Π, has the form $\Sigma_{\mathit{trans}(\Pi)}$, interpreted as a set of temporal nogoods over $\mathcal{A} \cup \mathit{Bd}(\Pi)$, and not as a set of (non-temporal) nogoods over $\mathcal{A} \cup \mathcal{A}' \cup \mathit{Bd}(\Pi)$.

Example 5.
The set Ψ_{Π_1} of temporal nogoods for Π_1 is $\{\{\mathbf{T}a', \mathbf{F}b\}, \{\mathbf{F}b', \mathbf{T}b\}, \{\mathbf{F}c', \mathbf{T}a\}, \{\mathbf{T}d', \mathbf{T}b\}, \{\mathbf{T}c, \mathbf{F}d\}, \{\mathbf{F}a', \mathbf{F}c\}, \{\mathbf{F}a', \mathbf{T}c', \mathbf{F}a\}\}$.

Temporal nogoods provide an alternative characterization of the nogoods of $\mathit{gen}(\Pi, n)$.

Proposition 1.
If Π is a temporal logic program and $n \geq 1$ then $\Sigma_{\mathit{gen}(\Pi,n)} = \Psi_\Pi[1, n]$.

In words, the nogoods for $\mathit{gen}(\Pi, n)$ are the same as the instantiation of the temporal nogoods for Π, that are nothing else than the nogoods of the logic program $\mathit{trans}(\Pi)$ interpreted as temporal nogoods.

Example 6.
The set of nogoods $\Sigma_{\mathit{gen}(\Pi_1,4)}$ is equal to $\Psi_{\Pi_1}[1, 4] = \bigcup_{i \in [1,4]} \{\{\mathbf{T}a_{i-1}, \mathbf{F}b_i\}, \{\mathbf{F}b_{i-1}, \mathbf{T}b_i\}, \{\mathbf{F}c_{i-1}, \mathbf{T}a_i\}, \{\mathbf{T}d_{i-1}, \mathbf{T}b_i\}, \{\mathbf{T}c_i, \mathbf{F}d_i\}, \{\mathbf{F}a_{i-1}, \mathbf{F}c_i\}, \{\mathbf{F}a_{i-1}, \mathbf{T}c_{i-1}, \mathbf{F}a_i\}\}$.

By Theorem 1, the temporal nogoods can be used to characterize the solutions of temporal logic problems.

Theorem 2.
Let (Π, I, F) be a temporal logic problem over $\mathcal{A}$. The pair (X, n) is a solution to (Π, I, F) for $n \geq 1$ and $X \subseteq \mathcal{A}[0, n]$ iff $X = S^{\mathbf{T}} \cap \mathcal{A}[0, n]$ for a (unique) solution S for $\Psi_\Pi[1, n]$ such that $I[0] \cup F[n] \subseteq S$.

5 Generalization of learned constraints

A common software architecture to solve a temporal problem (Π, I, F) combines a scheduler that assigns resources to different values of n, with one or many solvers that look for solutions of the assigned lengths n (see (Rintanen *et al.* 2006). The standard approach for the solvers is to extend the program $gen(\Pi, n)$ with facts and integrity constraints to adequately represent I and F and call the procedure *CDNL-ASP* with this extended program without assumptions. However, as we have seen in our introductory example, this method does not work well for our purposes, because it leads to a nogood representation of the initial and the final steps that is different from the nogood representation of the other steps. Hence, the constraints learned using nogoods specific to the initial and final steps may not be generalizable to the other steps. To overcome this issue, in our approach the solvers apply the procedure *CDNL-ASP*$(gen(\Pi, n), I[0] \cup F[n])$ to the generator program for Π and n, using assumptions to fix the assignments about the initial and final situations. Observe that in this case, by Proposition 1, the solver initially contains exactly the nogoods $\Psi_\Pi[1, n]$, and all the nogoods that it learns afterward are resolvents of $\Psi_\Pi[1, n]$.

Once this is settled, we ask ourselves:

> *What generalizations of the nogoods learned by CDNL-ASP can be applied to the same or other problems?*

We make the question more precise step by step. First, instead of talking about "the nogoods learned by the algorithm", we refer to the resolvents of $\Psi_\Pi[1, n]$ for some temporal problem (Π, I, F). Or more precisely, we refer to the resolvents of $\Psi_\Pi[i, j]$ for some i and j such that $1 \leq i \leq j \leq n$, since the learned nogoods are always the result of resolving nogoods belonging to some interval $[i, j]$ that may be smaller than $[1, n]$.

To formalize the notion of the "generalizations of nogoods", we introduce some notation for shifting a non-temporal nogood an amount of t time steps. For integers $n \geq 1$ and t, and a non-temporal nogood δ over $(\mathcal{A} \cup \mathcal{A}' \cup Bd(\Pi))[1, n]$, the symbol $\delta\langle t \rangle$ denotes the nogood that results from replacing in δ any signed literal $\mathbf{T}\alpha_m$ ($\mathbf{F}\alpha_m$) by $\mathbf{T}\alpha_{m+t}$ (by $\mathbf{F}\alpha_{m+t}$, respectively). For example, $\delta\langle 0 \rangle = \delta$, and if $\delta = \{\mathbf{T}a_2, \mathbf{F}b_3\}$, then $\delta\langle 1 \rangle$ is $\{\mathbf{T}a_3, \mathbf{F}b_4\}$, and $\delta\langle -1 \rangle$ is $\{\mathbf{T}a_1, \mathbf{F}b_2\}$. We say that $\delta\langle t \rangle$ is a *shifted version* of the nogood δ, and that a *generalization* of a nogood is a set of some of its shifted versions. For example, $\{\{\mathbf{T}a_2, \mathbf{F}b_3\}\}$ and $\{\{\mathbf{T}a_1, \mathbf{F}b_2\}, \{\mathbf{T}a_2, \mathbf{F}b_3\}, \{\mathbf{T}a_3, \mathbf{F}b_4\}\}$ are generalizations of $\{\mathbf{T}a_2, \mathbf{F}b_3\}$ and of $\{\mathbf{T}a_3, \mathbf{F}b_4\}$.

Next, by the "other problems" mentioned in the question, we refer to variations m of the length of the solution and to variations (Π, I', F') of the original problem where the initial and final situation may change, but the temporal program remains the same. Then, a generalization of a nogood "can be applied" to such problems if it can be added to the

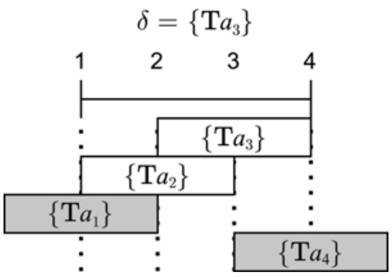

Fig. 1. Representation of different shifted versions of the nogood $\delta = \{\mathbf{T}a_3\}$. The surrounding rectangles cover the interval of the nogoods needed to prove them. For example, the rectangle of $\{\mathbf{T}a_2\}$ covers the interval $[1, 3]$ because $\{\mathbf{T}a_2\}$ is a resolvent of $\Psi_{\Pi_1}[1, 3]$.

set of nogoods used by the algorithm *CDNL-ASP* without changing the solutions to the problem. For any variation (Π, I', F'), those nogoods are $\Psi_\Pi[1, m]$, and a generalization can be added to them if the generalization is entailed by them. Hence, a generalization of a nogood "can be applied" to "some problem" (Π, I', F'), searching for a solution of length m, if the generalization is entailed by $\Psi_\Pi[1, m]$. Putting all together, we can rephrase our question as follows:

> *Given some temporal logic problem* (Π, I, F), *what generalizations of a resolvent* δ *of* $\Psi_\Pi[i, j]$ *are entailed by* $\Psi_\Pi[1, m]$?

Example 7.
Consider a call of $\mathit{CDNL\text{-}ASP}(\mathit{gen}(\Pi_1, 4), \emptyset)$ to search for a solution of length 4 to the temporal problem $(\Pi_1, \emptyset, \emptyset)$. Initially, the solver may choose to make a_3 true by adding $\mathbf{T}a_3$ to the initial assignment. Then, by unit propagation, it could derive the literal $\mathbf{T}c_2$ by $\{\mathbf{F}c_2, \mathbf{T}a_3\}$, the literal $\mathbf{T}d_2$ by $\{\mathbf{T}c_2, \mathbf{F}d_2\}$, the literal $\mathbf{F}b_3$ by $\{\mathbf{T}d_2, \mathbf{T}b_3\}$, and the literal $\mathbf{F}b_4$ by $\{\mathbf{F}b_3, \mathbf{T}b_4\}$, leading to a conflict due to the violation of the nogood $\{\mathbf{T}a_3, \mathbf{F}b_4\}$. At this stage, the solver would learn the nogood $\delta = \{\mathbf{T}a_3\}$ by resolving iteratively $\{\mathbf{T}a_3, \mathbf{F}b_4\}$ with the nogoods $\{\mathbf{F}b_3, \mathbf{T}b_4\}$, $\{\mathbf{T}d_2, \mathbf{T}b_3\}$, $\{\mathbf{T}c_2, \mathbf{F}d_2\}$, and $\{\mathbf{F}c_2, \mathbf{T}a_3\}$ used for propagation. Hence, δ is a resolvent of the set of those nogoods. Moreover, given that those nogoods are instantiations of some temporal nogoods of Ψ_{Π_1} at the interval $[2, 4]$, δ is also a resolvent of $\Psi_{\Pi_1}[2, 4]$ and of $\Psi_{\Pi_1}[1, 4]$. Observe that, by shifting the nogoods 1 time point backward, we obtain that $\delta\langle -1\rangle = \{\mathbf{T}a_2\}$ is a resolvent of $\Psi_{\Pi_1}[1, 3]$, and therefore also of $\Psi_{\Pi_1}[1, 4]$. Then, by the correctness of resolution, we have that the generalization $\{\{\mathbf{T}a_2\}, \{\mathbf{T}a_3\}\}$ of δ is entailed by $\Psi_{\Pi_1}[1, 4]$. On the other hand, $\delta\langle -2\rangle = \{\mathbf{T}a_1\}$ is a resolvent of $\Psi_{\Pi_1}[0, 2]$, but not of $\Psi_{\Pi_1}[1, 4]$, since the instantiations at 0 do not belong to $\Psi_{\Pi_1}[1, 4]$. Similarly, $\delta\langle 1\rangle = \{\mathbf{T}a_4\}$ is a resolvent of $\Psi_{\Pi_1}[3, 5]$, but not of $\Psi_{\Pi_1}[1, 4]$, since the instantations at 5 do not belong to $\Psi_{\Pi_1}[1, 4]$ (see Figure 1).

This example suggests a sufficient condition for the generalization of a nogood δ learned from $\Psi_\Pi[i, j]$: a shifted version $\delta\langle t\rangle$ of some generalization of δ is entailed by $\Psi_\Pi[1, n]$ if the nogoods that result from shifting $\Psi_\Pi[i, j]$ an amount of t time points belong to $\Psi_\Pi[1, n]$.

Theorem 3.
Let Π be a temporal logic program, and δ be a resolvent of $\Psi_\Pi[i,j]$ for some i and j such that $1 \leq i \leq j$. Then, for any $n \geq 1$, the set of nogoods $\Psi_\Pi[1,n]$ entails the generalization

$$\{\delta\langle t\rangle \mid [i+t, j+t] \subseteq [1,n]\}.$$

This theorem is based on the fact that the resolution proof that derived δ from $\Psi_\Pi[i,j]$ can be used to derive every $\delta\langle t\rangle$ from $\Psi_\Pi[i+t, j+t]$, simply by shifting the nogoods t time steps. This means that $\delta\langle t\rangle$ is a resolvent of $\Psi_\Pi[i+t, j+t]$. Given that $[i+t, j+t] \subseteq [1,n]$, the nogood $\delta\langle t\rangle$ is also a resolvent of $\Psi_\Pi[1,n]$. Then, the theorem follows from the correctness of resolution.

This result allows us to generalize the learned nogoods to different lengths and different initial and final situations. Following our example, if we were now searching for a solution of length 9 to the temporal problem $(\Pi_1, \{\mathbf{T}c\}, \{\mathbf{T}b\})$, we could add the generalization $\{\{\mathbf{T}a_i\} \mid i \in [2,8]\}$ to $\mathit{CDNL\text{-}ASP}(\mathit{gen}(\Pi_1, 9), \{\mathbf{T}c_0, \mathbf{T}b_9\})$. Observe that this method requires knowing the specific interval $[i,j]$ of the nogoods used to derive a learned constraint. To obtain this information in *clingo*, we would have to modify the implementation of the solving algorithm. We leave that option for future work, and in the next section we follow another approach that does not require to modify the solver.

6 When can we generalize all learned nogoods to all time steps?

Given *any temporal problem*, Theorem 3 gives us a sufficient condition for the generalization of the nogoods learned while solving that problem. In this section, we investigate for *what kind of temporal problems* can we generalize all learned nogoods to all time steps. In other words, we would like to know when can we add the generalization:

$$\{\delta\langle t\rangle \mid \mathit{step}(\delta\langle t\rangle) \subseteq [0,n]\}$$

of a learned nogood δ to the set of nogoods used by algorithm *CDNL-ASP*.

Example 8.
In Example 7, we saw that the nogood $\delta = \{\mathbf{T}a_3\}$ is a resolvent of $\Psi_{\Pi_1}[2,4]$. By Theorem 3, it follows that the generalization $\{\{\mathbf{T}a_2\}, \{\mathbf{T}a_3\}\}$ of δ is entailed by $\Psi_{\Pi_1}[1,n]$, but that theorem does not allow us to infer that $\delta\langle -2\rangle = \{\mathbf{T}a_1\}$ is entailed by $\Psi_{\Pi_1}[1,n]$. In fact, this would be incorrect since all the solutions to $\Psi_{\Pi_1}[1,n]$ contain the literal $\mathbf{T}a_1$. But why is $\{\mathbf{T}a_1\}$ not entailed by $\Psi_{\Pi_1}[1,n]$? One reason for this is that $\Psi_{\Pi_1}[1,n]$ does not entail the nogood $\{\mathbf{T}c_0, \mathbf{F}d_0\}$, that would be necessary to derive $\{\mathbf{T}a_1\}$. In fact, since the initial state of all solutions of length 4 to $(\Pi_1, \emptyset, \emptyset)$ is $\{a,b,c\}$, all solutions to $\Psi_{\Pi_1}[1,4]$ contain the literals $\mathbf{T}c_0$ and $\mathbf{F}d_0$, violating the nogood $\{\mathbf{T}c_0, \mathbf{F}d_0\}$. But this implies that the initial state $\{a,b,c\}$ cannot have some previous state. This is obvious looking at Π_1, since $\{a,b,c\}$ violates the integrity constraint $\bot \leftarrow c, \neg d$. On the other hand, if $\{a,b,c\}$ had some previous state, then it would not violate that integrity constraint, and therefore $\Psi_{\Pi_1}[1,n]$ would entail $\{\mathbf{T}c_0, \mathbf{F}d_0\}$ and then $\{\mathbf{T}a_1\}$.

This analysis suggests that we can always add a learned nogood, shifted backward in time, if the initial states occur in some path with sufficient states before them. A similar analysis in the other direction – for instance, for nogood $\delta\langle 1\rangle$ in the previous example –

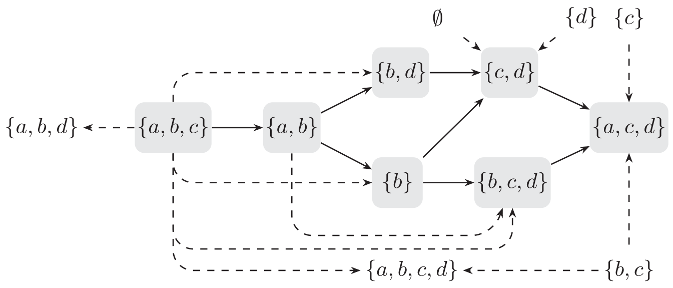

Fig. 2. Transition $G(\Pi_1)$ of temporal program Π_1. The nodes that belong to some solution of length 4 have a gray background. The transitions of those solutions are represented by normal arrows, while the other arrows are dashed.

suggests that we can add a learned nogood, shifted forward in time, if the final states occur in some path with sufficient states after them. These observations lead us to this informal answer: we can generalize all learned nogoods to all time steps if the initial states have enough preceding states and the final states have enough following states. We make this claim precise in the following.

In Section 4 we introduced transition programs and used them to characterize the solutions of a given temporal problem. Transition programs *trans*(Π) define transitions between the states of some temporal program Π. In turn, these transitions implicitly define a *transition* graph $G(\Pi)$. Formally, given a temporal logic program Π over $\mathcal{A}$, the transition graph $G(\Pi)$ is the graph (N, E) where E is the set of edges

$$\{(\{a \mid a' \in X \cap \mathcal{A}'\}, X \cap \mathcal{A}) \mid X \text{ is a stable model of } \mathit{trans}(\Pi)\}$$

and N is the set of nodes occurring in some edge of E, that is, $N = \bigcup_{(X,Y)\in E}\{X, Y\}$. Every node in N is also a state of Π, so we may use both names interchangeably. A *path* of a transition graph (N, E) is a sequence of states $(X_0, \ldots, X_n)$ such that for every $i \in [1, n]$ the pair (X_{i-1}, X_i) is an edge from E. We say that the *length* of such a path is n. We usually denote the states occurring in a path by symbols of the form X_i, where the subindex i represents the position of the state in the path, and does *not* represent the instantiation of some set X to time step i, which would be represented by $X[i]$.

We can characterize the solutions to a temporal problem (Π, I, F) as the finite paths of $G(\Pi)$ whose first and final nodes are consistent with I and F, respectively. We state this formally in the next theorem, that extends Theorem 2.

Theorem 4.
Let (Π, I, F) be a temporal logic problem over $\mathcal{A}$, $n \geq 1$, and X be a set of atoms over $\mathcal{A}[0, n]$. Then, the following statements are equivalent:

- The pair (X, n) is a solution to (Π, I, F).
- $X = S^{\mathbf{T}} \cap \mathcal{A}[0, n]$ for a solution S for $\Psi_\Pi[1, n]$ such that $I[0] \cup F[n] \subseteq S$.
- There is a path $(X_0, \ldots, X_n)$ in $G(\Pi)$ such that $X = \bigcup_{i\in[0,n]} X_i[i]$, the state X_0 is consistent with I, and the state X_n is consistent with F.

Figure 2 represents the transition $G(\Pi_1)$ of temporal program Π_1. There are only three paths in $G(\Pi)$ of length 4:

- $(\{a,b,c\},\{a,b\},\{b,d\},\{c,d\},\{a,c,d\})$,
- $(\{a,b,c\},\{a,b\},\{b\},\{c,d\},\{a,c,d\})$, and
- $(\{a,b,c\},\{a,b\},\{b\},\{b,c,d\},\{a,c,d\})$.

By Theorem 4, each of them corresponds to one of the solutions to $(\Pi_1,\emptyset,\emptyset)$ of length 4.

Theorem 4 establishes a relation between the solutions to the set of nogoods $\Psi_\Pi[1,n]$ and the paths of $G(\Pi)$. This leads naturally to a relation between the nogoods entailed by $\Psi_\Pi[1,n]$ and the paths of $G(\Pi)$.

Example 9.
We have seen that the set of nogoods $\Psi_{\Pi_1}[2,4]$ entails the nogood $\delta=\{\mathbf{T}a_3\}$. It follows that $\Psi_{\Pi_1}[1,3]$ entails $\delta\langle -1\rangle=\{\mathbf{T}a_2\}$. By Theorem 4 we have that the solutions to $\Psi_{\Pi_1}[1,3]$ correspond to the paths (X_0,X_1,X_2,X_3) in $G(\Pi_1)$. Then, if the solutions to $\Psi_{\Pi_1}[1,3]$ do not violate the nogood $\{\mathbf{T}a_2\}$, it follows that the paths (X_0,X_1,X_2,X_3) in $G(\Pi_1)$ do not violate this nogood either –we make precise this relation between nogoods and paths below. Hence, the fact that $\Psi_{\Pi_1}[2,4]$ entails $\{\mathbf{T}a_3\}$ means that, in every path (X_0,X_1,X_2,X_3) in $G(\Pi_1)$, the state X_2 cannot include the atom a. We can check this in $G(\Pi_1)$, where the only states that appear in such a position are $\{b,d\}$, $\{c,d\}$, $\{b\}$ and $\{b,c,d\}$, and neither of them contains a. On the other hand, the states $\{a,b,c\}$ and $\{a,b\}$ can contain a because they are not in the third position of any path, and the same holds for the final state $\{a,c,d\}$, since it does not occur in the penultimate position of any path.

We formalize the relation between nogoods and paths. For simplicity, we only discuss the case where learned nogoods consist of normal atoms, but the extension to body atoms does not pose any special challenge, since body atoms can be seen as a conjunction of literals over normal atoms. Let Π be a temporal program over $\mathcal{A}$, let $(X_0,\dots,X_n)$ be some path in $G(\Pi)$, and δ be some (non-temporal) nogood over $\mathcal{A}[0,n]$. We say that the path $(X_0,\dots,X_n)$ violates δ if

$$\delta\subseteq\bigcup_{i\in[0,n]}(\{\mathbf{T}a_i\mid a\in X_i\}\cup\{\mathbf{F}a_i\mid a\in\mathcal{A}\setminus X_i\}).$$

The right-hand side of the equation represents the path as an assignment.

Proposition 2.
Let Π be a temporal logic program over $\mathcal{A}$, $n\geq 1$, and let δ be a (non-temporal) nogood over $\mathcal{A}[0,n]$. Then, the following two statements are equivalent:

- The set of nogoods $\Psi_\Pi[1,n]$ entails δ.
- Every path $(X_0,\dots,X_n)$ of length n in $G(\Pi)$ does not violate δ.

Our next theorem answers *formally* the question of this section. To introduce it, we define a node in a given graph as a *source* if it has no incoming edges, as a *sink* if it has no outgoing edges, and as an *internal* node otherwise. Then, we say that a temporal program Π is *internal* if every node in $G(\Pi)$ is internal. The theorem states that we can generalize all learned nogoods to all time steps when the temporal program is *internal*.

Observe that if Π is internal, then every path can be extended arbitrarily from both ends. This means that every state has always "enough preceding states" and "enough following states," as we said earlier.

Proposition 3.
Let Π be a internal temporal program. If $(X_0, \dots, X_n)$ is a path of length n in $G(\Pi)$, then for any $i, j \geq 0$ there is a path $(Y_0, \dots, Y_{n+i+j})$ of length $n+i+j$ in $G(\Pi)$ such that $(X_0, \dots, X_n) = (Y_i, \dots, Y_{n+i})$.

Example 10.
Program Π_1 is not internal. For example, the node $\{a, b, c\}$ has no incoming edge, and the node $\{a, c, d\}$ has no outgoing edge.

Example 11.
Let Π_2 the program that results from replacing in Π_1 the rules $\bot \leftarrow c, \neg d$ and $\bot \leftarrow \neg a', c', \neg a$ by $\bot \leftarrow b, d$ and $\bot \leftarrow c, \neg b$. One can check that program Π_2 is cyclic, and that the nogood $\{\mathbf{F}b_1, \mathbf{F}a_2\}$ is a resolvent of $\Psi_{\Pi_2}[2,3]$. By Theorem 3, we can conclude that $\{\mathbf{F}b_0, \mathbf{F}a_1\}$ and $\{\mathbf{F}b_1, \mathbf{F}a_2\}$ are entailed by $\Psi_{\Pi_2}[1,3]$, but we cannot do the same about $\{\mathbf{F}b_2, \mathbf{F}a_3\}$. On the other hand, by Proposition 2, the fact that $\Psi_{\Pi_2}[1,3]$ entails $\{\mathbf{F}b_1, \mathbf{F}a_2\}$ implies that every path $(X_0, \dots, X_3)$ of length 3 in $G(\Pi_2)$ does not violate $\{\mathbf{F}b_1, \mathbf{F}a_2\}$. Given that Π is internal, it follows that every path (Y_0, Y_1) of length 1 in $G(\Pi_2)$ does not violate $\{\mathbf{F}b_0, \mathbf{F}a_1\}$. Otherwise, by Proposition 3 there would be some path $(X_0, \dots, X_3)$ of length 3 in $G(\Pi)$ such that $(Y_0, Y_1) = (X_1, X_2)$, and this path would violate $\{\mathbf{F}b_1, \mathbf{F}a_2\}$ – which would be a contradiction. Given that the paths (Y_0, Y_1) in $G(\Pi_2)$ do not violate $\{\mathbf{F}b_0, \mathbf{F}a_1\}$, Proposition 2 implies that $\Psi_{\Pi_2}[1,1]$ entails $\{\mathbf{F}b_0, \mathbf{F}a_1\}$. Therefore, $\Psi_{\Pi_2}[i,i]$ entails $\{\mathbf{F}b_{i-1}, \mathbf{F}a_i\}$ for $i \in [1,4]$, and $\Psi_{\Pi_2}[1,4]$ entails all those nogoods together. In other words, $\Psi_{\Pi_2}[1,4]$ entails all shifted versions of $\{\mathbf{F}b_1, \mathbf{F}a_2\}$ that fit in the interval $[0,4]$.

Theorem 5.
Let Π be a temporal logic program, and δ be a resolvent of $\Psi_\Pi[i,j]$ for $1 \leq i \leq j$. If Π is internal, for any $n \geq 1$, the set of nogoods $\Psi_\Pi[1,n]$ entails the generalization

$$\{\delta\langle t\rangle \mid \mathit{step}(\delta\langle t\rangle) \subseteq [0,n]\}.$$

Theorem 5 allows the generalization of learned nogoods to all time steps when the temporal program is internal. Unfortunately, internal temporal programs are not very useful for representing planning problems. To see this, consider some planning problem and any of its transitions, where an action a in state X results in state Y. If actions occur at the same time point as their effects – as in our introductory example – the transition graph must have an edge between X and $Y \cup \{a\}$. Since actions at the previous step can be chosen freely, for any action b there will also be an edge between $X \cup \{b\}$ and $Y \cup \{a\}$. But if the temporal program is internal, then $X \cup \{b\}$ must have an incoming edge. Hence, X must result from executing action b in some state. Overall, this means that every state X that leads to another state Y must be the result of *every possible* action under some previous state, which clearly does not apply to many planning problems. Something similar happens if action occurrences are placed before their effects.

To address this issue, we extend our study by taking into account the initial states associated with temporal logic problems. We refine the definition of being internal by considering only the initial states and the states that are reachable from them. This modification resolves the problem observed in our previous example: if the state $X \cup \{b\}$ has no incoming edges, then it is not reachable from any initial state, and no condition is imposed on it.

Let Π be a temporal logic program and I be a partial assignment. A state X of Π is

- *initial wrt I* if it is consistent with I and has some outgoing edge in $G(\Pi)$;
- *reachable wrt I* if it can be reached from some initial state of Π wrt I, that is if there is some path $(Y, \dots, X)$ in $G(\Pi)$ starting at some initial state Y of Π wrt I; and
- *loop-reachable* if it can be reached from some loop in $G(\Pi)$, that is if there is some path of the form $(Y, \dots, Y, \dots, X)$ in $G(\Pi)$ for some state Y of Π.

The temporal program Π is *internal wrt I* if these conditions hold:

(i) Every initial state wrt I is loop-reachable.
(ii) Every reachable state wrt I is internal.

Condition (i) guarantees that initial states have "enough previous states", and condition (ii) guarantees that reachable states have "enough successor states". Together, they imply that any path starting at some initial or reachable state can be extended indefinitely in both directions.

Proposition 4.
Let Π be a temporal logic program and I be a partial assignment such that Π is internal wrt I. If $(X_0, \dots, X_n)$ is a path of length n in $G(\Pi)$ and X_0 is initial or reachable wrt I, then for any $i, j \geq 0$ there is a path $(Y_0, \dots, Y_{n+i+j})$ of length $n+i+j$ in $G(\Pi)$ such that $(X_0, \dots, X_n) = (Y_i, \dots, Y_{n+i})$.

Temporal programs that are internal with respect to a partial assignment generalize the notion of internal temporal programs.

Proposition 5.
A temporal logic program Π is internal iff it is internal wrt the empty assignment.

If a temporal program is internal wrt an assignment I, we can generalize all learned nogoods to all time steps provided that I *holds initially*. We enforce this condition by adding the set of nogoods $\{\{\mathbf{F}a_0\} \mid \mathbf{T}a \in I\} \cup \{\{\mathbf{T}a_0\} \mid \mathbf{F}a \in I\}$ that we denote by *nogoods*(I).

Theorem 6.
Let Π be a temporal logic program, I be a partial assignment, and δ be a resolvent of $\Psi_\Pi[i,j]$ for $1 \leq i \leq j$. If Π is internal wrt I, then for any $n \geq 1$, the set of nogoods $\Psi_\Pi[1,n] \cup \mathit{nogoods}(I)$ entails the generalization

$$\{\delta\langle t\rangle \mid \mathit{step}(\delta\langle t\rangle) \subseteq [0,n]\}.$$

In theory, the inclusion of $nogoods(I)$ appears to conflict with our goal of applying the learned nogoods to variations of the original problem with different initial states. But in practice this need not pose a problem, since the assignment I may reflect a form of representing problems in general – such as placing actions at the step where their effects occur – and be independent of any specific instance. As a result, the learned nogoods may remain applicable to all intended variations of the original problem.

This observation holds for most representations of planning problems that we have encountered, which can be captured by temporal logic programs internal wrt a generic assignment I. Such an assignment ensures that

(a) initial states have no action occurrences.

To achieve this, actions must occur at the same time step as their effects. In ASP, this is usually a matter of convenience. To satisfy condition (i) of being internal wrt I, it suffices if:

(b) whenever no action occurs in a state, that state remains unchanged.

Conditions (a) and (b) create a loop in all initial states, making them loop-reachable. To meet condition (b), the representation should allow the non-execution of actions, and the law of inertia should make all fluents persist. Most representations of planning problem conform to this, or can be adapted with minor changes. Condition (ii) of being internal wrt I can be satisfied directly if after the execution of each action, another action can always be applied. An alternative and more generic condition is the following:

(c) from every state with an action occurrence, there is a transition to the same state without action occurrences.

In addition to the representation of inertia, this condition only requires that the representation always allows the non-execution of actions. Based on our experience, most representations of planning problems satisfy conditions (a-c) or, as we have seen in our introductory example, can be easily modified to do that. In fact, the encodings used in our conference paper (Romero *et al.* 2022) required only minor changes to fit within this approach.

In summary, our methodology represents a planning problem as a temporal logic problem (Π, I, F) such that the assignment I is divided into:

- A generic assignment I_1 that ensures condition (a).
- A specific assignment I_2 that describes the initial situation of the planning problem.

Moreover, the temporal program Π is construed to satisfy conditions (b) and (c), ensuring that it is internal wrt I_1. Given this, to solve (Π, I, F) we run the procedure $\mathit{CDNL\text{-}ASP}(gen(\Pi, n), I_1[0] \cup I_2[0] \cup F[n])$ for different lengths n. For each run, any nogood δ learned by the solver is a resolvent of $\Psi_\Pi[1, n]$. By Theorem 6, we can generalize δ across all time steps, also in other runs with different lengths, and in other variations of the original problem. Such variations can be represented as temporal logic problems $(\Pi, I_1 \cup I_2', F')$ where Π and I_1 remain unchanged. As before, we can solve them using $\mathit{CDNL\text{-}ASP}(gen(\Pi, n), I_1[0] \cup {I_2}'[0] \cup F'[n])$ for different lengths n. Since the solutions computed by such calls are solutions of $\Psi_\Pi[1,n] \cup \mathit{nogoods}(I_1)$, by Theorem 6 the generalization of δ to all time steps can also be added to them.

7 From non-internal to internal temporal programs

If we want to solve the temporal problem $(\Pi_1, \emptyset, \emptyset)$ and generalize the nogoods learned during this process, the results from the previous section are not applicable because Π_1 is not internal, and the initial assignment is empty. To address this limitation, we introduce a method to translate any temporal program into an internal one that preserves the same solutions modulo the original atoms. We can solve the original problem using the new program, and given that this program is internal, we can generalize all learned nogoods to all time steps. Moreover, we can apply these nogoods to the original program, after eliminating the auxiliary atoms introduced by the translation.

Let Π be a temporal logic program over $\mathcal{A}$, and let λ be a fresh atom not included in $\mathcal{A}$. We define $\lambda(\Pi)$ as the temporal logic program

$$\{\{\lambda\} \leftarrow\} \cup \tag{9}$$

$$\{h(r) \leftarrow Bd(r) \cup \{\lambda\} \mid r \in \Pi\} \cup \tag{10}$$

$$\{\{a\} \leftarrow \neg\lambda \mid a \in \mathcal{A}\}. \tag{11}$$

This translation extends Π by introducing a choice rule for λ, tagging the rules of Π with λ, and allowing the selection of any subset of $\mathcal{A}$ when λ does not hold.

To understand the translation, let us study the relationship between the transition graphs defined by Π and $\lambda(\Pi)$. These graphs are defined by the corresponding transition programs:

- $\mathit{trans}(\Pi) = \Pi \cup \mathit{Choice}(\mathcal{A}')$, and
- $\mathit{trans}(\lambda(\Pi)) = \lambda(\Pi) \cup \mathit{Choice}(\mathcal{A}' \cup \{\lambda'\})$.

The behavior of $\mathit{trans}(\lambda(\Pi))$ depends on whether λ is chosen at (9) or not. If λ is chosen, then $\mathit{trans}(\lambda(\Pi))$ has the same stable models as the program

$$\{\lambda \leftarrow\} \cup \mathit{trans}(\Pi) \cup \{\{\lambda'\} \leftarrow\}$$

as the rules in (10) can be simplified to the original rules from Π, and the rules in (11) become irrelevant. Then, for every stable model X of $\mathit{trans}(\Pi)$, the sets $X \cup \{\lambda\}$ and $X \cup \{\lambda, \lambda'\}$ are stable models of $\mathit{trans}(\lambda(\Pi))$. In terms of transition graphs, this means that for every edge (X, Y) in $G(\Pi)$ there are two corresponding edges $(X, Y \cup \{\lambda\})$ and $(X \cup \{\lambda\}, Y \cup \{\lambda\})$ in $G(\lambda(\Pi))$.

If λ is not chosen, then $\mathit{trans}(\lambda(\Pi))$ has the same stable models as the program

$$\{\{a\} \leftarrow \mid a \in \mathcal{A}\} \cup \mathit{Choice}(\mathcal{A}' \cup \{\lambda'\})$$

as the rules in (10) have no effect, and the body of (11) becomes true. Then, for every $X, Y \subseteq \mathcal{A}$, if X' denotes the set $\{a' \mid a \in X\}$, the sets $X' \cup Y$ and $X' \cup Y \cup \{\lambda'\}$ are stable models of $\mathit{trans}(\lambda(\Pi))$. Accordingly, $G(\lambda(\Pi))$ contains the edges (X, Y) and $(X \cup \{\lambda\}, Y)$.

Proposition 6.
Let Π be a temporal logic program, and let E be the set of edges in its transition graph $G(\Pi)$. The set of edges in $G(\lambda(\Pi))$ is the union of the following four sets:

- $\{(X, Y \cup \{\lambda\}) \mid (X, Y) \in E\}$
- $\{(X \cup \{\lambda\}, Y \cup \{\lambda\}) \mid (X, Y) \in E\}$
- $\{(X, Y) \mid X, Y \subseteq \mathcal{A}\}$
- $\{(X \cup \{\lambda\}, Y) \mid X, Y \subseteq \mathcal{A}\}$

Given these sets, particularly the second, there is a one-to-one correspondence between the paths in $G(\Pi)$ and the paths in $G(\lambda(\Pi))$ where λ belongs to all states. This correspondence holds even if λ does not occur in the initial state of the path, since the first edge can also be taken from the first set of the previous proposition. By Theorem 4, this correspondence between paths induces a correspondence between the solutions to Π and the solutions to $\lambda(\Pi)$ where λ is true always after the initial step. To formalize this, we say that a solution (X, n) to a given temporal problem is λ-*normal* if $\lambda_0 \notin X$ and $\lambda_i \in X$ for all $i \in [1, n]$.

Proposition 7.
Let (Π, I, F) be a temporal logic problem. There is a one-to-one correspondence between the solutions to (Π, I, F) and the λ-normal solutions to $(\lambda(\Pi), I, F)$.

Example 12.
The temporal problem $(\Pi_1, \emptyset, \emptyset)$ has three solutions of length 4: $(X, 4)$, $(X \cup \{d_2\}, 4)$, and $(X \cup \{b_3\}, 4)$, where X is defined as in Example 3. These solutions correspond one-to-one to the three λ-normal solutions of the same length to $(\lambda(\Pi_1), \emptyset, \emptyset)$: $(X \cup Y, 4)$, $(X \cup \{d_2\} \cup Y, 4)$, and $(X \cup \{b_3\} \cup Y, 4)$, where Y is $\{\lambda_1, \lambda_2, \lambda_3, \lambda_4\}$.

The call $\mathit{CDNL\text{-}ASP}(\mathit{gen}(\lambda(\Pi), n),\ I[0] \cup F[n] \cup \{\mathbf{F}\lambda_0, \mathbf{T}\lambda_1, \ldots, \mathbf{T}\lambda_n\})$ computes λ-normal solutions to $(\lambda(\Pi), I, F)$, enforcing the correct value for λ at every time point using assumptions. The solutions to the original problem $(\lambda(\Pi), I, F)$ can be obtained from these λ-normal solutions by removing the atoms $\{\lambda_1, \ldots, \lambda_n\}$.

The following proposition establishes that the program $\lambda(\Pi)$ is internal wrt $\{\mathbf{F}\lambda_0\}$. To see why, observe that the third and fourth sets of Proposition 6 show that, in $G(\lambda(\Pi))$, every state X –whether it contains λ or not – is connected to the state $X \setminus \{\lambda\}$. This implies that:

(a) every state without λ is connected to itself, and
(b) every state has a successor state.

Condition (a) provides condition (i) for Π being internal wrt $\{\mathbf{F}\lambda_0\}$, since initial states do not have λ; and condition (b) ensures the corresponding condition (ii).

Proposition 8.
For any temporal program Π, the program $\lambda(\Pi)$ is internal wrt $\{\mathbf{F}\lambda_0\}$.

Since $\lambda(\Pi)$ is internal wrt $\{\mathbf{F}\lambda_0\}$, by Theorem 6 we obtain the next result, that allows us to generalize all nogoods learned using $\Psi_{\lambda(\Pi)}$ to all time steps, as long as λ is initially false.

Theorem 7.
Let Π be a temporal logic program, and δ be a resolvent of $\Psi_{\lambda(\Pi)}[i,j]$ for some i and j such that $1 \leq i \leq j$. For any $n \geq 1$, the set of nogoods $\Psi_{\lambda(\Pi)}[1,n] \cup \{\{\mathbf{T}\lambda_0\}\}$ entails the generalization

$$\{\delta\langle t\rangle \mid \mathit{step}(\delta\langle t\rangle) \subseteq [0,n]\}.$$

Example 13.
The set $\Psi_{\lambda(\Pi_1)}$ of temporal nogoods of $\lambda(\Pi)$ is $\{\delta \cup \{\mathbf{T}\lambda\} \mid \delta \in \Psi_{\Pi_1}\}$ (Ψ_{Π_1} is given in Example 5). We showed in Example 7 that $\{\mathbf{T}a_3\}$ is a resolvent of $\Psi_{\Pi_1}[2,4]$. Similarly, the nogood $\delta = \{\mathbf{T}a_3, \mathbf{T}\lambda_2, \mathbf{T}\lambda_3, \mathbf{T}\lambda_4\}$ is a resolvent of $\Psi_{\lambda(\Pi_1)}[2,4]$, derived by iteratively resolving $\{\mathbf{T}a_3, \mathbf{F}b_4, \mathbf{T}\lambda_4\}$ with $\{\mathbf{F}b_3, \mathbf{T}b_4, \mathbf{T}\lambda_4\}$, $\{\mathbf{T}d_2, \mathbf{T}b_3, \mathbf{T}\lambda_3\}$, $\{\mathbf{T}c_2, \mathbf{F}d_2, \mathbf{T}\lambda_2\}$, and $\{\mathbf{F}c_2, \mathbf{T}a_3, \mathbf{T}\lambda_3\}$. Note that the additional $\mathbf{T}\lambda_i$ literals in δ indicate the time steps of the nogoods used in the resolution proof. The nogood δ and its shifted versions $\delta\langle -1\rangle = \{\mathbf{T}a_2, \mathbf{T}\lambda_1, \mathbf{T}\lambda_2, \mathbf{T}\lambda_3\}$ and $\delta\langle -2\rangle = \{\mathbf{T}a_1, \mathbf{T}\lambda_0, \mathbf{T}\lambda_1, \mathbf{T}\lambda_2\}$ fall within the interval $[0,4]$. Hence, by Theorem 7, they are entailed by $\Psi_{\lambda(\Pi_1)}[1,4] \cup \{\mathbf{T}\lambda_0\}$. In particular, $\delta\langle -2\rangle$ is trivially entailed, since the solutions to $\Psi_{\lambda(\Pi_1)}[1,4] \cup \{\mathbf{T}\lambda_0\}$ cannot include $\mathbf{T}\lambda_0$, which belongs to $\delta\langle -2\rangle$.

The nogoods learned from the program $\lambda(\Pi)$ can be applied to the original program Π after removing all auxiliary atoms. We define the *simplification* of a nogood δ, denoted by $\mathit{simp}(\delta)$, as the nogood obtained from δ after removing all literals of the form $\mathbf{F}\lambda_i$ or $\mathbf{T}\lambda_i$. for any integer i.

Theorem 8.
Let Π be a temporal logic program, and let δ be a resolvent of $\Psi_{\lambda(\Pi)}[i,j]$ for some i and j such that $1 \leq i \leq j$. For any $n \geq 1$, the set of nogoods $\Psi_{\Pi}[1,n]$ entails the generalization

$$\{\mathit{simp}(\delta\langle t\rangle) \mid \mathit{step}(\delta\langle t\rangle) \subseteq [0,n], \mathbf{T}\lambda_0 \notin \delta\langle t\rangle\}.$$

Example 14.
The nogood $\delta = \{\mathbf{T}a_3, \mathbf{T}\lambda_2, \mathbf{T}\lambda_3, \mathbf{T}\lambda_4\}$ is a resolvent of $\Psi_{\lambda(\Pi_1)}[2,4]$. Both δ and its shifted versions $\delta\langle -1\rangle$ and $\delta\langle -2\rangle$ fall within the interval $[0,4]$, but $\delta\langle -2\rangle$ includes $\mathbf{T}\lambda_0$. Hence, by Theorem 8, the set of nogoods $\Psi_{\Pi_1}[1,4]$ entails $\mathit{simp}(\delta) = \{\mathbf{T}a_3\}$ and $\mathit{simp}(\delta\langle -1\rangle) = \{\mathbf{T}a_2\}$. Note that the condition $\mathbf{T}\lambda_0 \notin \delta\langle t\rangle$ of Theorem 8 is necessary, as without it, we would incorrectly infer that $\mathit{simp}(\delta\langle -2\rangle) = \{\mathbf{T}a_1\}$ is entailed by $\Psi_{\Pi_1}[1,4]$. In the end, we arrive at the same conclusion as in Example 7, but in that case our reasoning relied on prior knowledge – not available to the *CDNL-ASP* algorithm – about the time steps $[2,4]$ of the nogoods used to derive $\{\mathbf{T}a_3\}$, while in this case the information about the time steps is encoded directly in δ through the $\mathbf{T}\lambda_i$ literals, making this additional knowledge unnecessary.

8 Experiments

We experimentally evaluate the generalization of learned nogoods in ASP planning using the solver *clingo*. The goal of the experiments is to study the performance of *clingo* when the planning encodings are extended by the generalizations of some constraints learned by *clingo* itself. We are interested only in the solving time and not in the grounding time, but in any case we have observed no differences between grounding times among the different configurations. We did experiments in two different settings, single-shot and multi-shot, that we detail below. Following the approach of (Gebser *et al.* 2016), in all experiments we disregarded the learned nogoods of size greater than 50 and of degree greater than 10, where the degree of a nogood is defined as the difference between the maximum and minimum step of the literals of the nogood.[1] In all the experiments, the learned nogoods are always sorted either by size or by *literal block distance* (*lbd*, (Audemard and Simon, 2009)), a measure that is usually associated with the quality of a learned nogood. We tried configurations adding the best 500, 1000, or 1500 nogoods, according to either their nogood size or their *lbd*. The results ordering the nogoods by *lbd* were similar but slightly better than those ordering by size, and here we focus on them. We used two benchmark sets from (Dimopoulos *et al.* 2018). The first consists of PDDL benchmarks from planning competitions, translated to ASP using the system *plasp* presented in that paper. This set contains 120 instances of 6 different domains. The second set consists of ASP planning benchmarks from ASP competitions. It contains 136 instances of 9 domains. We adapted the logic programs of these benchmarks to the format of temporal logic programs as follows: we deleted the facts used to specify the initial situation, as well as the integrity constraints used to specify the goal, we added some choice rules to open the initial situation, and we fixed the initial situation and the goal using assumptions. All benchmarks were run using the version 5.5.1 of *clingo* on an Intel Xeon E5-2650v4 under Debian GNU/Linux 10, with a memory limit of 8 GB, and a timeout of 15 minutes per instance.

The task in the single-shot experiment is to find a plan of a fixed length n that is part of the input. For the PDDL benchmarks we consider plan lengths varying from 5 to 75 in steps of 5 units, for a total of 2040 instances. The ASP benchmarks already have a plan length, and we use it. In a preliminary learning step, *clingo* is run with every instance for 10 minutes or until 16000 nogoods are learned, whatever happens first. The actual learning time is disregarded and not taken into account in the tables. Some PDDL instances reach the memory limit in this phase[2] We leave them aside and are left with 1663 instances of this type. We compare the performance of *clingo* running normally (baseline), versus the (learning) configurations where we add the best 500, 1000, or 1500 learned nogoods according to their *lbd* value. In this case we apply Theorem 6 and learn the nogoods using a slight variation of the original encoding, but use the original encoding for the evaluation of all configurations.

Tables 1 and 2 show the results for the PDDL and the ASP benchmarks, respectively. The first columns include the name and number of instances of every domain. The tables show the average solving times and the number of timeouts, in parenthesis, for every configuration and domain. We can observe that in general the learning configurations are faster than the baseline and in some domains they solve more instances. The improvement is not huge, but is persistent among the different settings. The only exception is the *elevator* domain in PDDL, where the baseline is a bit faster than the other configurations.

[1] These values are experimentally chosen. The higher the size and degree of the nogoods, the less useful they are as they become more specific.

[2] Since the initial state is left open, the grounding size increases and may exceed the memory limit.

Table 1. *Single-shot solving of PDDL benchmarks*

		baseline	500	1000	1500
blocks	(300)	**0.5**(0)	0.6(0)	0.6(0)	0.6(0)
depots	(270)	146.8(30)	140.2(30)	**124.2**(24)	135.5(28)
driverlog	(135)	14.0(1)	13.5(1)	11.5(1)	**10.8**(1)
elevator	(300)	**3.0**(0)	5.1(0)	4.3(0)	5.2(0)
grid	(30)	11.4(0)	6.0(0)	4.4(0)	**3.7**(0)
gripper	(255)	381.1(96)	380.9(90)	**360.9**(87)	367.7(90)
logistics	(225)	**0.5**(0)	**0.5**(0)	**0.5**(0)	0.8(0)
mystery	(130)	79.6(6)	71.1(3)	**58.6**(4)	64.6(6)
Total	(1645)	91.5(133)	90.0(124)	**83.0**(116)	86.5(125)

Table 2. *Single-shot solving of ASP benchmarks*

		baseline	500	1000	1500
HanoiTower	(20)	160.5 (2)	139.5 (0)	**137.9** (0)	143.9 (1)
Labyrinth	(20)	**246.6** (3)	348.3 (5)	284.8 (4)	296.2 (5)
Nomistery	(20)	585.7 (12)	545.4 (11)	**510.2** (9)	566.7 (12)
Ricochet robots	(20)	464.5 (9)	**320.3** (3)	410.8 (6)	404.9 (5)
Sokoban	(20)	458.7 (9)	454.2 (9)	453.8 (9)	**446.3** (9)
Visit-all	(20)	**559.1** (12)	562.5 (12)	560.7 (12)	561.5 (12)
Total	(120)	412.5 (47)	395.0 (40)	**393.0** (40)	403.3 (44)

We also analyzed the average number of conflicts per domain and configuration, and the results follow the same trend as the solving times.

In the multi-shot solving experiment, the solver first looks for a plan of length 5. If the solver finds no such plan, then it looks for a plan of length 10, and so on until it finds a plan. At each of these solver calls, we collect the best learned nogoods. Then, before the next solver call, we add the generalization of the best 500, 1000, or 1500 of them, depending on the configuration. As before, we rely on Theorem 6, but this time we use the same original encoding, slightly modified, for both learning and solving.[3] The results for PDDL and ASP are shown in Tables 3 and 4, respectively. In both of them, the baseline and the different configurations perform similarly, and there is not a clear trend. The analysis of the number of conflicts shows similar results.

We expected similar results in the single-shot and the multi-shot solving experiments, but the learning configurations outperformed the baseline in the former, while they performed similarly in the latter. We do not have a clear explanation for this, but we can suggest some hypotheses. In the single-shot experiments we can select the best nogood from a larger set than in the multi-shot experiments, which could influence the quality of the learned nogoods. In addition, having the learned constraints from the start

[3] The results using the translations from our conference paper (Romero *et al.* 2022) are similar (see the supplementary material).

Table 3. *Multi-shot solving of PDDL benchmarks*

		baseline	500	1000	1500
blocks	(20)	1.3 (0)	**0.7** (0)	**0.7** (0)	**0.7** (0)
depots	(18)	**148.6** (2)	255.9 (3)	188.7 (3)	221.9 (3)
driverlog	(9)	108.8 (1)	**102.1** (1)	104.8 (1)	108.6 (1)
elevator	(20)	**280.4** (5)	285.6 (5)	293.8 (5)	304.6 (5)
freecell	(16)	900.0 (16)	900.0 (16)	900.0 (16)	900.0 (16)
grid	(2)	5.1 (0)	**3.9** (0)	4.1 (0)	4.3 (0)
gripper	(17)	848.6 (16)	**847.5** (16)	849.0 (16)	847.9 (16)
logistics	(20)	**225.2** (5)	225.3 (5)	225.4 (5)	225.3 (5)
mystery	(14)	**321.8** (5)	321.9 (5)	321.9 (5)	321.9 (5)
Total	(136)	**346.6** (50)	360.9 (51)	353.6 (51)	359.7 (51)

Table 4. *Multi-shot solving of ASP benchmarks*

		baseline	500	1000	1500
HanoiTower	(20)	**440.8** (8)	512.8 (9)	489.4 (9)	498.9 (9)
Labyrinth	(20)	633.9 (14)	**633.8** (14)	**633.8** (14)	633.9 (14)
Nomistery	(20)	380.7 (7)	**363.1** (6)	381.0 (7)	384.7 (7)
Ricochet robots	(20)	**521.5** (11)	523.9 (11)	527.9 (11)	526.0 (11)
Sokoban	(20)	**721.5** (16)	**721.5** (16)	721.9 (16)	722.1 (16)
Visit-all	(20)	900.0 (20)	900.0 (20)	900.0 (20)	900.0 (20)
Total	(120)	**599.7** (76)	609.2 (76)	609.0 (77)	610.9 (77)

could positively influence the solver's heuristic. Moreover, the fixed plan length in the single-shot experiments could also improve the learning process.

9 Conclusion

CDCL is the key to the success of modern ASP solvers. So far, however, ASP solvers could not exploit the temporal structure of dynamic problems. We addressed this by elaborating upon the generalization of learned constraints in ASP solving for temporal domains. We started with the definition of temporal logic programs and problems. We studied the conditions under which learned constraints can be generalized, and we identified a class of temporal programs for which every learned nogood can be generalized to all time points. It turns out that many ASP planning encodings fall into this class, or can be easily adapted to it. We complemented this with a translation from temporal programs in general to temporal programs of that class. Our experimental evaluation show mixed results. In some settings, the addition of the learned constraints results in a consistent improvement of performance, while in others the performance is similar to the baseline. We plan to continue this experimental investigation in the future. Another avenue of future work is to continue the approach sketched at the end of Section 5 and develop a dedicated implementation within an ASP solver based on Theorem 3.

Acknowledgments

This work was supported by DFG grant SCHA 550/15. Thanks to Roland Kaminski and the anonymous referees for their useful comments.

Competing interests

The authors declare none.

Supplementary material

To view supplementary material for this article, please visit https://doi.org/10.1017/S1471068424000462.

References

Aguado, F., Cabalar, P., Diéguez, M., Pérez, G. and Vidal, C. 2013. Temporal equilibrium logic: a survey. *Journal of Applied Non-Classical Logics* 23, 1-2, 2–24.

Audemard, G. and Simon, L. 2009. Predicting learnt clauses quality in modern SAT solvers. In *Proc. of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09).* C. Boutilier, Ed. AAAI/MIT Press, 399–404

Cabalar, P., Diéguez, M., Hahn, S. and Schaub, T. 2021. Automata for dynamic answer set solving: Preliminary report. In *Proc. of the Fourteenth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'21).* Joaquín Arias, Fabio Aurelio D'Asaro, Abeer Dyoub, Gopal Gupta, Markus Hecher, Emily LeBlanc, Rafael Peñaloza, Elmer Salazar, Ari Saptawijaya, Felix Weitkämper and Jessica Zangari, Ed. CEUR-WS.org.

Cabalar, P., Kaminski, R., Morkisch, P. and Schaub, T. 2019. telingo = ASP + time. In *Proc. of the Fifteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'19).* M. Balduccini, Y. Lierler and S. Woltran, Eds. Lecture Notes in Artificial Intelligence, vol. 11481. Springer-Verlag, 256–269

Complai-taupe, R., Weinzierl, A. and Friedrich, G. 2020. Conflict generalisation in asp: Learning correct and effective non-ground constraints. *Theory and Practice of Logic Programming* 20, 799–814.

Cuteri, B., Dodaro, C., Ricca, F. and Schüller, P. 2020. Overcoming the grounding bottleneck due to constraints in ASP solving: Constraints become propagators. In *Proc. of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI'20).* C. Bessiere, Ed. ijcai.org, 1688–1694

Dimopoulos, Y., Gebser, M., Lühne, P., Romero, J. and Schaub, T. 2017. plasp 3: Towards effective ASP planning. In *Proc. of the Fourteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17).* M. Balduccini and T. Janhunen, Eds. Lecture Notes in Artificial Intelligence, vol. 10377. Springer-Verlag, 286–300

Dimopoulos, Y., Gebser, M., Lühne, P., Romero, J. and Schaub, T. 2018. plasp 3: Towards effective ASP planning. *Theory and Practice of Logic Programming* 19, 3, 477–504.

Dodaro, C., Mazzotta, G. and Ricca, F. (2023) Compilation of tight ASP programs. In *Frontiers in Artificial Intelligence and Applications*, ECAI, K. Gal, A. Nowé, G. J. Nalepa, R. Fairstein and R.Radulescu, Eds. vol. 372, IOS Press, 557–564

Eén, N. and Sörensson, N. 2003. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science* 89, 4, 543–560.

Gebser, M., Kaminski, R., Kaufmann, B., Lühne, P., Romero, J. and Schaub, T. (2016) Answer set solving with generalized learned constraints, *Technical Communications of the Thirty-Second International Conference On Logic Programming (ICLP'16)*, Carro, M. and King, A., 52-9, pp. 15, OpenAccess Series in Informatics (OASIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 1-9

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. (2012) *Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning.* Morgan and Claypool Publishers

Gebser, M., Kaufmann, B., Neumann, A. and Schaub, T. 2007. Conflict-driven answer set solving. In *Proc. of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07).* M. Veloso, Ed. AAAI/MIT Press, 386–392

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proc. of the Fifth International Conference and Symposium of Logic Programming (ICLP'88).* R. Kowalski and K. Bowen, Eds. MIT Press, 1070–1080

Gelfond, M. and Lifschitz, V. 1998. Action languages. *Electronic Transactions on Artificial Intelligence* 3, 6, 193–210.

Lee, J., Lifschitz, V. and Yang, F. 2013. Action language BC: Preliminary report. In *Proceedings of the Twenty-third International Joint Conference on Artificial Intelligence (IJCAI'13).* F. Rossi, Ed. IJCAI/AAAI Press, 983–989

Lefèvre, C., Béatrix, C., Stéphan, I. and Garcia, L. 2017. ASPeRiX, a first-order forward chaining approach for answer set computing. *Theory and Practice of Logic Programming* 17, 3, 266–310.

Lifschitz, V. and Turner, H. 1994. Splitting a logic program. (Proceedings of the Eleventh International Conference on Logic Programming, MIT Press, 23–37.

Mazzotta, G., Ricca, F. and Dodaro, C. (2022) Compilation of aggregates in ASP systems, *AAAI.* AAAI Press, pp. 5834–5841.

Mcdermott, D. (1998). PDDL — the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control

Palù, A. D., Dovier, A., Pontelli, E. and Rossi, G. 2009. GASP: Answer set programming with lazy grounding. *Fundamenta Informaticae* 96, 3, 297–322.

Rintanen, J., Heljanko, K. and Niemelä, I. 2006. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence* 170, 12-13, 1031–1080.

Romero, J., Schaub, T. and Strauch, K. 2022. On the generalization of learned constraints for ASP solving in temporal domains. Proceedings, G. Governatori and Turhan, A., Lecture Notes in Computer Science, Rules and Reasoning - 6th International Joint Conference on Rules and Reasoning, RuleML+RR. 2022, Springer, Berlin, Germany, 13752, 20–37, Lecture Notes in Computer Science, September 26-28, 2022

Weinzierl, A., Taupe, R. and Friedrich, G. 2020. Advancing lazy-grounding ASP solving techniques — restarts, phase saving, heuristics, and more. *Theory and Practice of Logic Programming* 20, 5, 609–624.