TPLP **20** (5): 625–640, 2020.
doi:10.1017/S147106842000037X

Managing caching strategies for stream reasoning with reinforcement learning

CARMINE DODARO[1], THOMAS EITER[2], PAUL OGRIS[3], and KONSTANTIN SCHEKOTIHIN[3]

[1] *Department of Mathematics and Computer Science, University of Calabria, Italy*
(*e-mail:* `dodaro@mat.unical.it`)
[2] *Institute of Logic and Computation, KBS Group, Vienna University of Technology, Austria,*
(*e-mail:* `eiter@kr.tuwien.ac.at`)
[3] *Alpen-Adria-Universität, Klagenfurt, Austria,*
(*e-mail:* `{paul.ogris,konstantin.schekotihin}@aau.at`)

submitted 10 August 2020; revised 11 August 2020; accepted 12 August 2020

Abstract

Efficient decision-making over continuously changing data is essential for many application domains such as cyber-physical systems, industry digitalization, etc. Modern stream reasoning frameworks allow one to model and solve various real-world problems using incremental and continuous evaluation of programs as new data arrives in the stream. Applied techniques use, e.g., Datalog-like materialization or truth maintenance algorithms to avoid costly re-computations, thus ensuring low latency and high throughput of a stream reasoner. However, the expressiveness of existing approaches is quite limited and, e.g., they cannot be used to encode problems with constraints, which often appear in practice. In this paper, we suggest a novel approach that uses the Conflict-Driven Constraint Learning (CDCL) to efficiently update legacy solutions by using intelligent management of learned constraints. In particular, we study the applicability of reinforcement learning to continuously assess the utility of learned constraints computed in previous invocations of the solving algorithm for the current one. Evaluations conducted on real-world reconfiguration problems show that providing a CDCL algorithm with relevant learned constraints from previous iterations results in significant performance improvements of the algorithm in stream reasoning scenarios.

KEYWORDS: Stream reasoning, ASP, Reinforcement learning, Caching strategies

1 Introduction

Stream reasoning is an emerging branch of AI connecting distributed systems, databases, machine learning, and knowledge representation and reasoning (KRR) to create complex decision-making frameworks that operate on continuously changing data. The recently proposed LARS (Logic-based Analytic Reasoning over Streams) framework (Beck et al. 2018) is rooted in logic programming, with reasoners that allow one to efficiently model and solve real-world problems.

To ensure high throughput and low latency of a decision-making system, stream reasoners use various techniques to reduce the impact of redundant computations while updating their internal state as new data arrives. LASER (Bazoobandi et al. 2017) uses a Datalog-like fixed-point materialization of restricted (plain) LARS formulas combined with specific annotations of rules to avoid unnecessary re-evaluation for subsequent portions of the data stream. Although LASER demonstrates very high performance in the evaluations, the expressiveness of the plain LARS programs handled, which is restricted to stratified negation, is rather limited. A larger fragment of plain LARS, which enables modeling of problems with multiple models but no constraints, is supported by TICKER (Beck et al. 2017). This stream reasoner uses a justification-based truth maintenance method (Doyle 1979) to first retract parts of previously computed decisions that are inconsistent with new data in the input stream and extend then the partial assignment obtained to a model. To provide full support of plain LARS without sacrificing much latency or throughput, Eiter et al. (2019) presented a distributed processing approach. In particular, their distributed reasoner uses stream stratification (Beck et al. 2018) to decompose LARS programs into subprograms that can be evaluated by different instances of an ASP solver in parallel.

On the one hand, applying full-fledged ASP solvers in stream reasoning settings provides an easy way to model and solve various problems in practice, but on the other hand poses a big challenge to their reasoning algorithms. The Conflict-Driven Constraint Learning (CDCL)-based algorithms used in Boolean satisfiability (SAT) (Silva and Sakallah 1996) and ASP (Alviano et al. 2013; Kaufmann et al. 2016) are not suitable for continuous operation over a long time. By design, they have no specific means to efficiently manage their internal state while solving sequences of instances appearing from an input stream. Instead, they are geared to find one or multiple solutions of a single problem instance per run, regardless of whether one-shot or incremental/multi-shot (Nadel and Ryvchin 2012; Alviano et al. 2015; Audemard and Simon 2018; Gebser et al. 2019) solving is used. In the latter mode, solvers incrementally construct a model for a sequence of instances, where the next one is added to those previously stored. Thus, if any of the accumulated instances is unsatisfiable, the unsatisfiability will prevail. To avoid this, modern solvers like OCLINGO (Gebser et al. 2012) or GLUCOSE (Audemard and Simon 2018) keep all constraints that are inconsistent with the current instance in memory, in a deactivated state. Similarly, in other logic programming paradigms, such as XSB Prolog, reasoning over streams can be implemented using incremental tabling, which can efficiently track atoms appearing in the data stream and update relevant cached goals (Swift and Warren 2012). This allows an incremental solver to keep all learned constraints and exploit this information to solve the instance obtained in the next iteration.

The main drawback of the incremental strategy in streaming scenarios, however, is that a solver might forget constraints learned during restarts that are relevant for solving the next instance. For instance, a typical application of stream reasoners for cyber-physical systems (CPS) is to monitor and reconfigure a system such that it can react suitably to changes in its environment. In such a scenario, a part of the CPS may go down due to a failure or for maintenance and then be up again after a while, i.e., the system is back to its normal state. Thus the reasoner must reconfigure multiple times and should not forget the constraints learned for the normal system state, as instances corresponding to it occur most often in the data stream.

To address these issues, we make in this paper the following contributions:

(1) We present a reinforcement learning approach aiming at the identification of learned constraints having the highest utility for the overall stream reasoning process. Depending on the value of the learning rate parameter, the learner can make different assumptions about the utility of learned constraints. Thus, given a learning rate close to 1, the learner assumes that data appearing in the stream at some point in time is closely related to the data appearing in the next time point. That is, any subsequent state of a CPS is highly related to a previous one. Therefore, data cached while solving a problem instance can help to solve the subsequent one. In turn, given a relatively small learning rate, the learner gets more skeptical and modifies its estimates very slowly. In the CPS case, this means, for instance, that the learner expects the system always to return to one of its most frequent operation states after all issues that occurred have been resolved.

(2) We present a method that can efficiently cache and manage data computed by a solving algorithm. For our reference implementation, we extended the WASP solver with functionality for data exchange with external caches and equipped it with an overgrounding approach derived from the work of Calimeri et al. (2019).

(3) We conduct an extensive evaluation and parameter tuning of the suggested approach using a version of the Partner Unit Problem (PUP) (Aschinger et al. 2011), which represents a continuous operation (reconfiguration) of various safety systems, and n-Queens Completion (Gent et al. 2017). The results show that the new approach significantly outperforms existing systems for plain LARS using ASP solvers, like (Eiter et al. 2019). Furthermore, they underline interesting links between the working of stream reasoners and ASP solvers considered, and may guide the development of future systems.

2 Preliminaries

LARS extends ASP with specific features for various stream reasoning problems (Beck et al. 2018). For instance, using *window functions* one can access parts of a stream such as all data that appeared in a given time interval or the last n tuples of the stream. Besides windows, LARS has *temporal modalities*: *(i)* the *at* operator $@_t$ where t is a time point, *(ii)* the *everywhere* operator $\Box$, and *(iii)* the *somewhere* operator $\Diamond$. Plain LARS programs were translated into ASP programs either natively using a ticked encoding (Beck et al. 2018), external predicates (Beck et al. 2017), or functions (Eiter et al. 2019). We thus consider in the sequel only techniques aiming at performance improvements of a continuously running ASP solver in stream reasoning applications.

Syntax. A normal ASP program Π is a finite set of rules of the form

$$a \leftarrow l_1, \ldots, l_n \tag{1}$$

where a is an atom (which may be absent) and $l_1, \ldots, l_n$ are literals for $n \geq 0$. An *atom* is an expression of the form $p(t_1, \ldots, t_k)$, where p is a predicate symbol and $t_1, \ldots, t_k$ are *terms*, i.e., either a *variable* or a *constant*. A *literal* l is either an atom a_i (positive) or its negation $\sim a_i$ (negative), where $\sim$ is *negation as failure*; the complement (opposite) of l is denoted by $\bar{l}$, and we let $\overline{L} = \{\bar{l} \mid l \in L\}$. An atom, a literal, or a rule is *ground*, if no variables appear in it. The grounding of a program Π is the set Π^G of all ground rules constructible from rules $r \in \Pi$ by substituting each variable in r with some constant appearing in Π.

Given a rule r of the form (1), the set $H(r) = \{a\}$ is the *head* and the set $B(r) = B^+(r) \cup B^-(r) = \{l_1, \ldots, l_n\}$ is the *body* of r, where $B^+(r)$ and $B^-(r)$ contain the positive and negative body literals, respectively. A rule r is a *fact* if $B(r) = \emptyset$ and a *constraint* if $H(r) = \emptyset$.

Semantics. The semantics of an ASP program Π is given for its ground instantiation Π^G. Let $\mathscr{A}$ be the set of all ground literals occurring in Π^G. An *interpretation* is a set $I \subseteq \mathscr{A} \cup \overline{\mathscr{A}}$ of literals that is *consistent*, i.e., $I \cap \bar{I} = \emptyset$; each literal $l \in I$ is true, each literal $l \in \bar{I}$ is false, and any other literal is undefined. An interpretation I is *total*, if $\mathscr{A} \subseteq I \cup \bar{I}$. An interpretation I *satisfies* a rule $r \in \Pi^G$, if $H(r) \subseteq I$ whenever $B(r) \subseteq I$. A *model* of Π^G is a total interpretation I satisfying each $r \in \Pi^G$; moreover, I is *stable* (an *answer set*), if I is a $\subseteq$-minimal model of the reduct $\{H(r) \leftarrow B^+(r) \mid r \in \Pi^G, B^-(r) \cap \bar{I} = \emptyset\}$ (Gelfond and Lifschitz 1988). Any answer set of Π^G is also an answer set of Π. By $AS(\Pi)$ we denote the set of answer sets of Π, which are those of Π^G.

Conflict-Driven Constraint Learning (CDCL). Modern ASP solvers compute answer sets using a CDCL-based algorithm (Kaufmann et al. 2016) as illustrated by Algorithm 1. The algorithm

Algorithm 1: FindAnswerSet

```
Input  : a ground program $\Pi^G$, a set of assumptions $A$, and a set of constraints $C$
Output: a tuple $(I, C)$, where $I$ is an answer set or incoherent, and $C$ is a set of constraints
1  $I \leftarrow \emptyset$    $\Pi^W \leftarrow \Pi^G \cup \{\leftarrow \overline{p} \mid p \in A\}$
2  $I \leftarrow$ Propagate($\Pi^W \cup C$, $I$)
3  if $I$ is consistent and total then return $(I, C)$
4  if $I$ is consistent then
5      $I \leftarrow$ RestartIfNeeded($I$)
6      $C \leftarrow$ DeleteConstraints($C$)
7      $I \leftarrow$ ChooseUndefinedLiteral($I$)
8  else
9      $r \leftarrow$ CreateConstraint($\Pi^W \cup C$, $I$)
10     $I \leftarrow$ RestoreConsistency($\Pi^W \cup C$, $I$)
11     $C \leftarrow C \cup \{r\}$
12     if $I$ is inconsistent then return (incoherent, $C$)
13 goto 2
```

takes as input a ground program Π^G, a set of assumptions literals A and a set of constraints C. The idea of the algorithm is to iteratively build an answer set $I \supseteq A$ of the program $\Pi^G \cup C$ resp. to prove that no such an answer set exists. To this end, I is initially set to $\emptyset$, and a working program Π^W initialized to $\Pi^G \cup C_A$, where C_A are constraints enforcing the truth of literals in A. Function `Propagate` (line 2) extends I with all literals that can be deterministically inferred. After propagation, three cases may occur:

(i) I is consistent and total. Then I and C are returned.

(ii) I is consistent but not total. Then the algorithm uses a heuristic strategy to decide whether the computation must restart from scratch, to explore different branches of the search tree (line 5), and whether to delete some constraints in C (line 6). `ChooseUndefinedLiteral` extends then I with an undefined literal ℓ (called branching literal, line 7) selects by some heuristics. A subsequent propagation step infers then the consequences of this choice.

(iii) I is inconsistent. Thus there is a conflict and I is analyzed. The reason for the conflict is modeled by a fresh constraint r computed by `CreateConstraint` function (line 9). Then the algorithm backtracks (i.e. choices and their consequences are undone) until the consistency of I is restored (line 10, often called *backjumping*) and r is added to C (line 11). If the conflict is unavoidable, i.e., the consistency of the interpretation cannot be restored, the algorithm terminates returning (*incoherent*, C).

Function `CreateConstraint` is crucial for the good performance of the algorithm. Indeed, it acquires information from conflicts and computes a constraint to avoid exploring the same search branch repeatedly. However, the number of constraints added might be exponential in the program size. Therefore, some of the learned constraints must be periodically deleted by the function `DeleteConstraints`. An important note is that the algorithm internally associates each literal $\ell \in I$ with a *decision level*, denoted $dl(\ell)$ and computed as follows. Let $maxdl(I) = 0$ if $I = \emptyset$, and $maxdl(I) = max(\{dl(\ell) \mid \ell \in I\})$ otherwise. Then, $dl(\ell) = 1 + maxdl(I)$ if ℓ is a branching literal, and $dl(\ell) = maxdl(I)$ otherwise. Decision levels are used by `CreateConstraint`. In fact, whenever a new constraint r is learned, a positive value called *Literals Blocks Distance (LBD)*

(Audemard and Simon 2009) is associated with it representing the number of different decision levels appearing in r. The function `DeleteConstraints` removes constraints with large LBD values since learned constraints with small LBD are viewed as important.

3 Management of caching strategies

Stream reasoning paradigms based on logic programming, such as LARS, convert all incoming data into atoms (literals) and forward them to reasoners. A single stream reasoner thus gets a sequence of atom sets appearing in a data stream over time. We assume that these sets are sampled from the same distribution, but the parameters of this distribution are unknown.

Example 1 In many technical systems the distribution of events quite often follows some kind of a power-law distribution, like the Pareto principle – 20% of components of cars fail in 80% of all cases. Thus, it was observed that e.g. faults in software (Adams 1984) or content requested by users in content-centric networks (Rossi and Rossini 2012) obey Zipf's law.

Finding solutions for complex problem instances might take considerable time. Therefore, ASP solvers apply various caching techniques designed to identify, store, and reuse different results obtained by its search algorithm. Most of the modern solvers apply two caching techniques: constraints learning and progress (phase) saving (Pipatsrisawat and Darwiche 2007).

3.1 Constraint learning

As discussed in the previous section, the solver analyses every conflict found to learn a constraint preventing its reoccurrence in subsequent steps. During solving, the algorithm might learn many constraints and, as previous experiments show, their number might grow very fast and thus negatively impact its performance (Gomes et al. 1998; Huang 2007; Audemard and Simon 2018). Modern solvers, therefore, adopt various restart strategies that drop unimportant constraints.

Similarly to standard solving algorithms, the preservation of learned constraints might improve the performance of stream reasoners, as they provide valuable information about conflicts found during previous calls, as it happens e.g. in Assumption-based Truth Maintenance Systems (ATMS) (de Kleer 1986). The latter also record constraints by analyzing conflicts found during the search. The constraints are stored in a specific database and help the reasoner to determine whether a set of new assumptions or assignments contains a known contradiction. As a result, ATMS can significantly speed up repeated reasoning tasks. The main problem of ATMS is that it stores all constraints and can only drop those subsumed by recent constraints. Modern incremental solvers instead freeze constraints unused by the solver and reactivate them when needed (Audemard and Simon 2018). The decision – if a constraint must be frozen or activated – is usually made by a heuristic. Audemard and Simon used a progress saving measure defined by $|\mathscr{P} \cap r|$, where $r \in C$ is a learned constraint and $\mathscr{P}$ is a set of literals stored by progress saving, as discussed in the next section.

Modern stream reasoners use learned constraints only for one reasoning cycle, i.e. call of Alg. 1. For instance, TICKER (Beck et al. 2017) and the distributed reasoner (Eiter et al. 2019) apply ASP solvers to find answer streams for new incoming data. This approach, which we call RESTART, creates a new instance of an ASP solver each time reasoning is invoked. Specifically, it rewrites a given LARS program P into an ASP program Π. When new data appears in the input stream at time t, the reasoning process registers a set $L = L_t^+ \cup L_t^-$ of ground atoms, where L_t^+

and L_t^- comprise atoms that appeared in resp. disappeared from the stream. The set L is used to extend Π with facts and obtain a ground program Π^G. Finally, Alg. 1 is run to find answer sets of Π^G which correspond to the answer stream of the

A stream reasoner can store the constraints learned now and reuse them later. However, this might lead to increased memory consumption and decrease the propagation performance. Applying techniques from incremental solving, such as freezing/reactivating constraints directly, can be problematic. Their heuristics are geared to incremental answer set finding for one program, which may result from multiple grounding steps. In stream reasoning, Alg. 1 aims to find answer sets of different but possibly very similar ground programs for sets of atoms (dis)appearing in the input stream.

Heuristics. Reinforcement learning (RL) can be applied to finding required heuristics using various methods (Sutton and Barto 2018), which, in general, can be split into *model-based* or *model-free* methods. The former methods assume that the learning agent has multiple states and transition probabilities between these states as reactions on the actions of a learner are known. In the case of the stream reasoning, the states might correspond to sets of learned constraints active in the solver and transition probabilities to the likelihood that the current set will be replaced by another one when new data will appear in the stream. The model-free approaches do not make such assumptions. In this work, we focus on the latter since the development of models is quite complicated and often cannot be done automatically. Next, the learning methods are differentiated wrt. rewards. The *immediate reward* methods assume that a learner gets rewards after each action, whereas in the case of *delayed rewards* the learner gets feedback describing its success after a sequence of actions. We assume that immediate rewards are more suitable for stream reasoning because the utility of a previously learned constraint for the current call of Alg. 1 is available as soon as it terminates.

The most known learning problem with immediate rewards is the *multi-armed bandit* (Sutton and Barto 2018). In this problem, a learner has to select one out of k available actions aiming to maximize the expected total reward over some time period. The reward is sampled from some unknown probability distribution that depends on the chosen action. However, in our case the learner should select a subset of actions, where each action represents a learned constraint that must be unfrozen in the solver. Therefore, we are focusing on a *multi-armed bandit problem with multiple plays* (Anantharam et al. 1987), which can be formulated as: given a set $N = \{n_1, \dots, n_k\}$ of random variables with unknown means $\Theta = \{\theta_i = \mathbb{E}[n_i] \mid n_i \in N\}$ that are i.i.d. over time, at each time point t a set $N_t \subseteq N$ is selected according to weights W_{t-1} associated with the variables in N. The selected variables N_t are observed at t and a reward vector R_t is determined for them, which helps to compute a new weight set W_t that better approximates Θ.

In the context of stream reasoning, the random variables N correspond to the set C of learned constraints in Alg. 1. appearing in the input stream is unknown to the reasoner, but we stationary. Any atom set L_t generated by Alg. 1 processes Π_t^G and returns a pair (I_t, C_t), where C_t is a set of learned constraints. Every constraint $c \in C_t$ is associated with a reward $R_t(c)$, which depends on whether Alg. 1 used c for computing answer sets (positive) or not (negative reward). The learning algorithm uses these rewards to update its estimate w_t^c of the expected reward for unfreezing c in the solver. The goals is to find a set W of estimates of expected rewards for all known learned constraints, called *policy*, that maximizes the (weighted) sum of rewards at all time points when Alg. 1 is run; i.e., a policy should maximize the probability to select and activate a subset of constraints learned at times $1, \dots, t$ for propagation while finding answer sets

Similarly to Gai et al. (2012), we use an action-value method that for each constraint $c \in C$ determines its weight w_t at a time point t with an update rule:

$$w^c_{t+1} = w^c_t + \lambda \cdot [R_t(c) - w^c_t]$$

where $R_t(c)$ is a reward from activating/freezing constraint c at the time t and $0 < \lambda \leq 1$ is a constant determining the learning rate. For a constant learning rate, the update rule can be reformulated in a non-recursive form:

$$w^c_{t+1} = (1-\lambda)w^c_t + \lambda R_t(c) = (1-\lambda)((1-\lambda)w^c_{t-1} + \lambda R_{t-1}(c)) + \lambda R_t(c) =$$
$$= (1-\lambda)^2 w^c_{t-1} + (1-\lambda)\lambda R_{t-1}(c) + \lambda R_t(c) = \cdots = (1-\lambda)^t w^c_1 + \lambda \sum_{i=1}^{t} (1-\lambda)^{t-i} R_i.$$

Consequently, the learning method focuses on the latter rewards and gives increasingly higher discounts for old rewards. As a result, the longer a learned constraint is not used by the solver, the higher is the likelihood that the learner will advise the solver to delete it during the next restart.

Depending on the definition of the reward function, we can obtain different estimates W of the optimal policy W^*. In this paper, we consider the following reward function:

$$R_t(c) = a \cdot [1 - 2 \cdot LBD_t(c) + ua_t(c) - uf_t(c) - 0.25 \cdot nf_t(c)]$$

where *(i)* $LBD_t(c)$ is the value of the LBD heuristic computed by Alg. 1 for a learned constraint c; *(ii)* a is a coefficient selected wrt. the number of decision levels of a ground program, with $a = 20$ in the experiments; *(iii)* $uf_t(c) = 1$ if c was frozen, i.e. not initially provided to Alg. 1, but rediscovered during its execution; *(iv)* $ua_t(c) = 1$ if a constraint was provided to Alg. 1 and used by it; and *(v)* $nf_t(c) = 1$ if the constraint was frozen and not rediscovered. The coefficient nf allows a learner to penalize and subsequently remove constraints that were frozen for a long period of time.

Finally, we use the optimistic initial values (Sutton 1995) as the exploration strategy. That is, a learner as formulated above uses estimates of expected rewards to decide which constraints must be unfrozen in the solver before each call to Alg. 1. By specifying initial values w^c_1 much larger than the possible values of the reward function, we encourage the learner to use newly found constraints more often in order to determine good estimates for their expected rewards. For instance, if a constraint is learned at a higher decision level, it might get a high LBD value in the first reward. As a result, this constraint will never be unfrozen again. Optimistic initial values of w^c_1 allow the learner to avoid such cases.

3.2 *Progress saving*

Progress saving is a caching technique that stores assignments by the search algorithm to avoid recomputations caused by backjumping (non-chronological backtracking) (Gaschnig 1979). As practice shows, the latter may cause the deletion of assignments unrelated to found conflicts. Thus solutions of subproblems might be computed again. Progress saving can avoid this by keeping an array of literals deleted during the backjumping and using it for branching decisions.

The effect of progress saving is similar to the one of JTMS techniques (Doyle 1979) used in the TICKER reasoner (Beck et al. 2015). The latter labels each rule with a time interval in which it should be considered by the reasoner, which determines the interval by analyzing the rule body at whenever new data appear in the input stream. When a rule is activated, it is materialized and moved to a cache. At the same time, all rules with expired labels are removed. Just as the progress saving technique, TICKER always stores the model from the last reasoning iteration. At each reasoning call, TICKER updates the model by removing all literals derived from removed rules and adding new literals via propagating activated rules. Implementations of Alg. 1, e.g. WASP, can store the cache of progress saving over multiple calls. When underlying assumptions change, WASP can use this cache to restore the last model if the new assumptions are satisfied.

3.3 Implementation details

The discussed caching strategy based on reinforcement learning was implemented as shown in Alg. 2. Prior to executing the main algorithm, an overgrounded program Π^G is generated. Just as suggested by Calimeri et al. (2019), we disable all rule simplifications of a grounder and then provide it with sets of possible facts corresponding to all ground atoms possible in the input data stream. For reconfiguration problems that we consider in this paper, the required set of possible constants is finite and it corresponds to the set of CPS components. The resulting ground program includes all rules that can be generated for all possible instances of the given problem encoding. As our experience shows, a complete overgrounding performed by our method by default allows a solver to run all simplification, preprocessing, and other routines at once. This enables time savings when new data appears in the stream. The original approach of Calimeri et al. (2019) can however be considered when a complete overgrounding is too large. Especially this approach might be interesting in applications where decisions must be made depending on values measured by CPS sensors or computed by other subsystems.

Alg. 2 starts a solver and does overgrounding during initialization which may take longer than the grounding step of RESTART. During operation, Alg. 2 identifies new assumptions using the ground atoms from the input stream and determines k constraints to be activated in the solver. Next, it calls Alg. 1 that finds answer sets as well as statistics required to compute the rewards. Finally, the weights of the learned constraints are updated and a new portion of ground atoms is read from the stream. In our experiments, we kept the cache size small – $k = 3000$ and $n = 2 \cdot k$ – to ensure efficient execution of the propagation in Alg. 1.

4 Evaluation

The evaluation of the suggested approach for various learning rates was performed on two (re)configuration problems: the partner unit problem (PUP) (Aschinger et al. 2011) and n-Queens completion (QC) (Gent et al. 2017). We selected these problems because of the following reasons. First, (re)configuration – known also as self-healing or resilience – is an important topic widely discussed in the domain of cyber-physical systems (CPS), see e.g. (Hehenberger et al. 2016; Ratasich et al. 2019). Second, both problems can easily be encoded in plain LARS but, at the moment, they can only be solved using methods like RESTART, which translate LARS encodings into ASP. Finally, they are well-known in the community and were used as benchmarks in ASP Competitions.[1]

[1] `https://asparagus.cs.uni-potsdam.de/contest/` or `https://www.mat.unical.it/aspcomp2014/`

Algorithm 2: ProcessStream

```
input      : a ground program $\Pi^G$
parameters: learning rate $\lambda$, a number $n$ of constraints to store, and a number $k$ of
            constraints to activate
output     : an answer set $I$ or incoherent
global     : constraints set $C = \emptyset$, assumptions set $A = \emptyset$, list of weights $w = []$
1  $(L^+, L^-) \leftarrow$ ReadStream ()
2  $A \leftarrow$ UpdateAssumptions ($L^+$, $L^-$, $A$)
3  $C^w \leftarrow$ DescendingSort ($C$, $w$)           // Order constraints in $C$ w.r.t. $w$
4  $C_{use} \leftarrow$ From ($C^w$, 0, $k$)                // Use first $k$ constraints
5  $C_{frozen} \leftarrow$ From ($C^w$, $k$, $n$)             // Freeze $n-k$ constraints
6  $C \leftarrow C \setminus$ From ($C^w$, $n$, $|C^w|$)       // Remove remaining constraints
7  $(I, C) \leftarrow$ FindAnswerSet ($\Pi^G$, $A$, $C_{use}$)
8  $C \leftarrow C \cup C_{frozen}$
9  WriteStream ($I$)
10 for $c \in C$ do
11     $r \leftarrow$ ComputeReward ($c$)
12     if $c \notin (C_{use} \cup C_{frozen})$ then $w[c] \leftarrow w_1 + \lambda \cdot r$   // New constraint is learned
13     else $w[c] \leftarrow w[c] + \lambda \cdot (r - w[c])$   // Update constraint cumulative reward
14 goto 1
```

4.1 Partner Unit Problem

PUP is an abstract configuration problem with numerous industrial applications such as railroad interlocking systems, security monitoring systems, or peer-to-peer networks (Aschinger et al. 2011). For instance, in security monitoring applications, the goal is to ensure that at any time only an allowed number of persons are in each *security zone*. The movements of persons between rooms is registered by *sensors* that are placed on doors between the rooms as well as the entrances to the building. To ensure the security of the building, sensor readings and zone equipment must be connected over a network of *communication units*, where each unit has an equal number of "sensor" and "zone" ports defined by *Unit CAPacity* (*uCap*). A unit-to-unit network is established using one or more communication ports, whose number is defined by *Inter-Unit CAPacity* (*iuCap*). To ensure near real-time communication, it is required that if a zone/sensor is connected to a unit U, then all sensors/zones related to it must be attached either to U or to other units directly connected to U.

Example 2 (PUP security application) Consider a small PUP problem in which a building has four rooms and two entrances, shown in Fig. 1. It has six security zones controlling both entrances and all doors. To guarantee the security of the building, each zone should be able to read observations of door sensors registering movement of persons, e.g. the switch zone z_{123} comprises sensors s_1, s_2, and s_3 which control all three tracks of the switch. Zones sensors, and zone-to-sensor relations can be represented as a bipartite graph, shown in Fig. 1.

Formally, the partner unit problem can be defined as follows.

Definition 1 (PUP problem) *Let $P = \langle Z, S, E, U, uCap, iuCap \rangle$, where Z and S are sets of zones resp. sensors, $E \subseteq Z \times S$ is a set of zone-to-sensors relations, and U is a set of units with uCap*

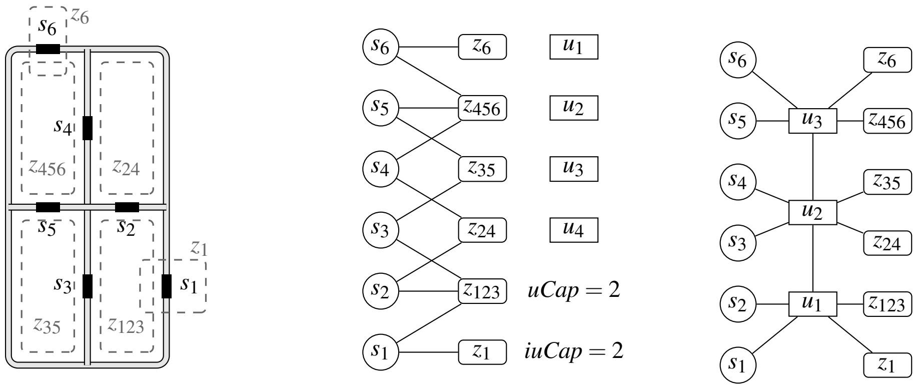

Fig. 1. Sample security monitoring system layout with the row length $n = 2$ (right), a derived PUP instance (center), and one of its solutions (right).

many zone/sensor ports and iuCap many inter-unit ports. A solution is a graph $L = \langle Z \cup S \cup U, H \rangle$ *with edges* $H \subseteq (Z \times U) \cup (S \times U) \cup (U \times U)$ *representing zone-to-unit, sensor-to-unit, and unit-to-unit relations such that*

1. *Each zone and sensor is connected to exactly one unit;*
2. *Each unit is connected to at most uCap zones/sensors resp. iuCap units (called partner units);*
3. *If a zone z and a sensor s are connected to different units, i.e.,* $(z,u),(s,u') \in H$ *where* $u \neq u'$*, then* $(z,s) \in E$ *implies* $(u,u') \in H$*.*

Example 3 (PUP security application, cont.) A possible solution graph for our example PUP problem instance is shown in Fig. 1. This solution uses three units that form a simple network allowing for the fast communication between door sensors and related security zones. ■

PUP in stream reasoning applications. Various security and safety applications of CPSs can be represented as PUP. Stream reasoning in these applications is used to monitor and (re)configure a CPS e.g. in case of administration actions, failures of system components, etc. For instance, an administrator may temporarily change a configuration of security zones for some event, a door sensor may fail, or a security zone can be deactivated for building maintenance. In such situations, changes in a CPS are continuously communicated to a stream reasoner, which should find a new configuration of the CPS for its new state. Solutions of new instances must be communicated back to the CPS as fast as possible to ensure the best results.

Instance generation. To simulate a stream of events from a CPS, we implemented a generator that applies random modifications to a given PUP instance, representing a CPS for a security monitoring application. The instance is selected from the family of *double* PUP instances representing security monitoring systems (Aschinger et al. 2011). It consists of two rows of rooms arranged on a grid and connected by doors wherever two rooms meet as in Fig. 1. The sensors are installed at each door and the security zones are laid over the rooms. We measure the instance size by a row length n, i.e., there are $2n$ zones and $3n-2$ sensors. In our evaluation all experiments were performed on instances with row lengths $n \in \{6, \dots, 11\}$.

The instances are modified by applying *mutation operators* that represent events registered by a monitoring component of a CPS stream reasoning system. Such monitoring can easily be done by LARS-based solutions as shown, e.g., in (Beck et al. 2017; Eiter et al. 2019). All such events are encoded by a set of ground atoms representing modifications to the current PUP instance.

To model real-world scenarios appearing in a security CPS, we use the following mutations: ($m1$) disabling a random zone, ($m2$) disabling a random sensor, ($m3$) restoring the original problem. Mutations $m1$ and $m2$ correspond to rooms out of order resp. doors becoming blocked and represent faults in the system, while mutation $m3$ corresponds to restoring the initial CPS state.

The generator uses randomization to simulate the randomly occurring events in the CPS. For the mutations $m1$ and $m2$, we assume that the involved zones and sensors are selected according to Zipf's law, which is often observed in practice when some components of a CPS are more likely to fail than the others (cf. Example 1). The mutation $m3$ is activated according to a Bernoulli distribution with a probability p.

To ensure the repeatability of each experiment for different algorithms and thus the comparability of results, the generator first creates a random ordering $e_1, \dots, e_n$ of the zones/sensors and builds a probability distribution such that the frequency of e_i is inverse proportional to the rank i. That is, the probability to select e_i is proportional to $1/i^{\alpha}$, where the parameter α controls the skew of the distribution towards $e_1, \dots, e_{i-1}$. For each experiment, we selected two values $\alpha \in \{2.2, 0.7\}$. These values ensure that $x\%$ of the elements are picked $(100-x)\%$ of the time, for $x \in \{20, 40\}$. For instance, for $\alpha = 2.2$ the sampling follows the Pareto Principle, by which 20% of the components fail in 80% of all cases. In general, the smaller the value of α the closer is Zipf's law to the uniform distribution. Note that the selected α values were specifically determined for our experiments to approximate the target selection ratios as close as possible.

Experiment. The goal of the experiment was to evaluate the performance of Alg. 2 while tackling various situations occurring in a CPS as well as the impact of the caching strategies. We generated a set of stream-reasoning instances using the mutation schema $[m1, m3, m2, m3]$ where $m3$ was applied with $p = 0.8$, i.e., there is an ≈80% chance that all modifications by previous mutations will be repaired. These settings resulted in PUP instances with ≈3 modifications. Each CPS simulation has a sequence of 256 PUP instances generated by repeating the mutation schema.

Since the optimal learning rate λ is unknown, we conducted a number of experiments with different λ-values, which results are shown in Fig. 2. The RESTART method has the worst performance, while the RL method needs for all values of λ comparable time to find a solution for each incoming PUP instance. As it turned out during the experiment, a relatively small subset of all learned constraints (mostly binary or ternary) had a dramatic impact on the performance of the solver. All learning strategies could identify those constraints and always unfreeze them in the solver. Therefore, in all RL experiments the reasoning time has a very small variance as the span of the 1st and the 3rd quartiles, indicated by boxes, as well as of the min and the max values, shown by whiskers, is very small. The topmost outlier, represented by a point, corresponds to the solving time of the first instance, which comprises the overgrounding time. However, the overgrounding is executed only once when the stream reasoning system is starting up and thus has no influence on the reasoning performance during operation time. The other outliers for smaller learning rates $\lambda \leq 0.1$ occurred because the learner was too conservative and could not react fast enough to data changes in the stream. This behavior can be explained by the selected exploration strategy. The learners with $\lambda > 0.1$ were able to update the optimistic initial weights of learned constraint to good estimates of expected rewards faster than the learners with $\lambda \leq 0.1$. Further experiments indicate that RL can solve streaming PUP instances up to row length 20 with the same median time as RESTART for row length 11. Moreover, RESTART spent most of the time for model search and not for program grounding, as shown in Table 1.

Furthermore, we performed experiments isolating each caching strategy to measure its impact

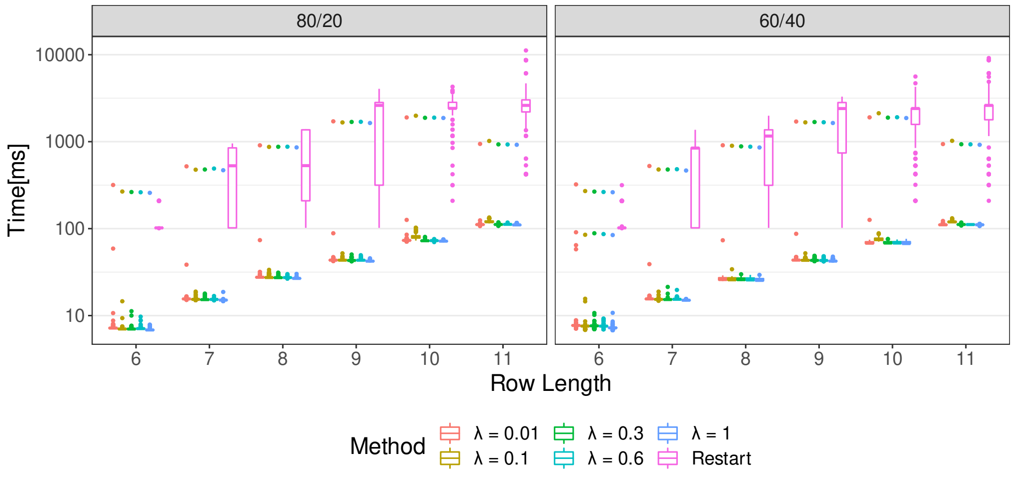

Fig. 2. Results of the experiment with the partner unit problem (PUP)

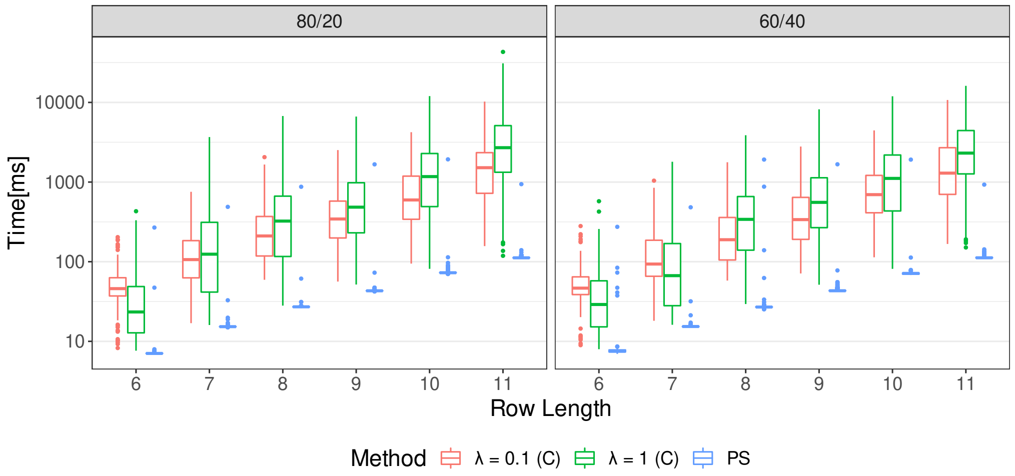

Fig. 3. Impact of the individual caching strategies in the PUP experiment

on the performance of the stream reasoner. In this experiment, RL used only constraints managed according to their strategies, labeled with (C) in Fig. 3, whereas PS keeps all constraints frozen thus forcing the solver to rely only on progress saving. The most interesting results were obtained for RL with $\lambda = 1$ and $\lambda = 0.1$. They indicate that progress saving has the largest impact on the

Row length	6	7	8	9	10	11
Restart	9.99	12.73	16.02	20.50	25.73	32.62
RL with $\lambda = 0.1$	28.04	47.40	77.13	122.80	187.55	266.35

Table 1. *Mean grounding times measured in milliseconds for the PUP instances*

reasoner performance. As consecutive instances are in this scenario very similar, only small parts of a legacy configuration become incoherent wrt. incoming facts. Progress saving (PS) allows the solver to rapidly reconstruct coherent parts of the model and then focus solely on the repair of its small incoherent part. However, the increased number of outliers indicates that in some cases repairs were not easy to find compared to RL (see Fig. 2). Among the "constraint-only" strategies $\lambda = 0.1$ appears to be a better learning rate on most of the instances. The performance of the learner initialized with $\lambda = 1$ degrades in line with the skewness of the distribution (cases 80/20 and 60/40) that allows for more different changes in an incoming instance. As this learner prefers constraints relevant to the model for the initial system state, it cannot select proper constraints if a rare event occurs.

We also made a similar experiment with $\alpha = 3.64$ where 10% of the components fail in 90% of all cases, and an experiment with single modifications, by setting $p = 1$ in the generator. The results are quite similar to those for multiple modifications (see online appendix).[2]

4.2 n-Queens Completion

The n-Queens Completion problem (QC) is well-known to be NP-complete (Gent et al. 2017) and an interesting benchmark for stream reasoning systems (Eiter et al. 2019), defined as follows: *given* an $n \times n$ chessboard and a set $Q = \{q_1, \ldots, q_k\}$, $k < n$, of queens q_i placed on it, such that no $q_i \neq q_j \in Q$ attack each other, *place* all remaining $n - |Q|$ queens on the board with that property.

Instance generation. The instance generator was implemented in a similar way as for PUP. First, it creates an initial QC instance by generating a set Q of $\lfloor 0.4 \cdot n \rfloor$ many queens which are randomly positioned on the board according to Corollary 15 (Gent et al. 2017), which guarantees the generation of a satisfiable QC instance. The instance is modified using the following mutations: ($m1$) rotate a board counterclockwise at 90°; ($m2$) place a random non-attacking queen on the board; and ($m3$) restore the original problem. The column in which a queen is added by $m2$ is selected wrt. Zipf's law, where our experiment used $\alpha = 1.35$. This value corresponds to a rather moderate skewness of the underlying distribution that leads to a more uniform selection of a column where a queen is placed. A row for a new queen was computed according to Corollary 15 where valid placements were evaluated in random order. Finally, the value of the restart probability for $m3$ was defined as $p = 0.95$. The generator was then used to create streaming test instances by iteratively applying the mutation schema $[m1, m2, m1, m3]$. Note that according to Gent et al. (2017), this generator is not a sustainable one, i.e., that can be used to find really difficult instances for modern solvers. Nevertheless, given the performance requirements to stream reasoners, the resulting instances are suitable for our experiments.

Experiment. We generated four streaming instances for $n \in \{14, 18, 22, 26, 30\}$ with 256 QC instances each and evaluated them for the same learning rates as in the previous section. The results, shown in Fig. 4, are that as in the previous experiment the performance of RL with different learning rates is comparable for the same reason and significantly outperforms RESTART: in the largest experiment, the worst RL result of ≈ 80 ms is more than 120 times better than the solving time of RESTART. Also, we can observe that for the large instances $n > 18$ the performance of learners that is comparable for the most optimistic $\lambda = 1$ and the most conservative $\lambda = 0.01$ learning rates. The remaining learning rates prevented the learner to stay focused on the most

[2] `http://distributed-stream-reasoner.ainf.at`

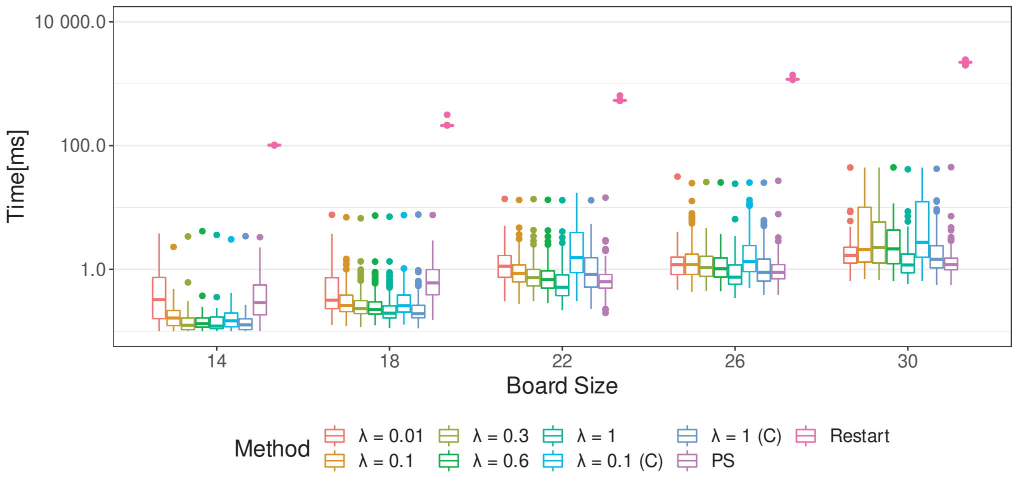

Fig. 4. Results of the experiment with the n-Queens Completion problem

useful constraints. However, the experiment also showed that the impact of individual caching strategies depends on the initial placement of queens. Thus, for the first two instances, the modifications introduced by $m1$ were large and the progress saving method (PS in Fig. 4) was unable to help the solver to reconstruct the solution. The learned clauses reused by RL without progress saving, labeled with (C) in Fig. 4, provided more information to the propagation algorithm for smaller instances with $n \leq 18$. Nevertheless, PS turned out to be useful while solving the larger instances with $n \geq 22$. In general, the experiment shows that it is quite hard to predict which learning rate would be more useful during the solving process. An extensive study of applying different learning strategies for balancing of various caching strategies remains for future work.

5 Conclusion

In this paper, we have discussed caching techniques used in modern ASP solvers and presented two approaches to the management of learned constraints based on reinforcement learning. The evaluation results that we presented indicate that proper reuse of data obtained while solving one instance from a data stream can significantly improve the performance of modern solvers while solving subsequent instances, and hence of ASP-based stream reasoning engines on top of them.

Moreover, the experiments provided support for the findings of previous research on the application of truth maintenance system techniques in stream reasoners like TICKER (Beck et al. 2017). Progress saving – a JTMS-like caching strategy – appears to be very useful in monitoring applications of technical systems. Since massive changes or failures are rarely observed in such environments under normal conditions, data delivered by such systems in subsequent time points is highly interrelated. Therefore, a solver using progress saving can easily restore consistent parts of a previous model and focus only on repairing of a rather small number of remaining unsatisfiable assignments. However, current progress saving methods are less versatile in comparison to JTMS techniques when the number of possible constants in a program is large. In such situations, they usually cannot select which literals must be removed from the cache as it grows in size. This finding opens an interesting direction of future research, especially in conjunction with predictive overgrounding techniques (Calimeri et al. 2019).

The positive effect of caching of learned constraints was observed in situations when data in the input stream caused many inconsistent assignments in the existing model. In such situations learned constraints were able to provide valuable information to the solver that could be fruitfully used to reduce its search time. However, our findings also showed that the application of reinforcement learning in this area must be studied in more detail. In our future work, we are going to focus on automated identification of reward functions that work best for a particular encoding in the LARS language and we shall consider experiments with highly dynamic domains, such as cooperative intelligent transport systems.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments that helped us to improve the paper a lot. This work was conducted in the scope of the research project *DynaCon (FFG-PNr.: 861263)* funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the program "ICT of the Future" between 2017 and 2020 (see `https://iktderzukunft.at/en/` for more information).

References

ADAMS, E. N. 1984. Optimizing preventive service of software products. *IBM J. Res. Dev. 28,* 1, 2–14.

ALVIANO, M., DODARO, C., FABER, W., LEONE, N., AND RICCA, F. 2013. WASP: A native ASP solver based on constraint learning. In *LPNMR*. 54–66.

ALVIANO, M., DODARO, C., LEONE, N., AND RICCA, F. 2015. Advances in WASP. In *LPNMR*. 40–54.

ANANTHARAM, V., VARAIYA, P., AND WALRAND, J. 1987. Asymptotically efficient allocation rules for the multiarmed bandit problem with multiple plays-Part I: I.I.D. rewards. *IEEE Trans. on Automatic Control 32,* 11, 968–976.

ASCHINGER, M., DRESCHER, C., FRIEDRICH, G., GOTTLOB, G., JEAVONS, P., RYABOKON, A., AND THORSTENSEN, E. 2011. Optimization methods for the partner units problem. In *CPAIOR*. 4–19.

AUDEMARD, G. AND SIMON, L. 2009. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*. 399–404.

AUDEMARD, G. AND SIMON, L. 2018. On the glucose SAT solver. *Int. J. Artif. Intell. Tools 27,* 1, 1–25.

BAZOOBANDI, H. R., BECK, H., AND URBANI, J. 2017. Expressive stream reasoning with laser. In *ISWC*. 87–103.

BECK, H., BIERBAUMER, B., DAO-TRAN, M., EITER, T., HELLWAGNER, H., AND SCHEKOTIHIN, K. 2017. Stream reasoning-based control of caching strategies in CCN routers. In *ICC*. IEEE, 1–6.

BECK, H., DAO-TRAN, M., AND EITER, T. 2015. Answer update for rule-based stream reasoning. In *IJCAI*. AAAI Press, 2741–2747.

BECK, H., DAO-TRAN, M., AND EITER, T. 2018. LARS: A logic-based framework for analytic reasoning over streams. *Artif. Intell. 261*, 16–70.

BECK, H., EITER, T., AND FOLIE, C. 2017. Ticker: A system for incremental asp-based stream reasoning. *TPLP 17,* 5-6, 744–763.

CALIMERI, F., IANNI, G., PACENZA, F., PERRI, S., AND ZANGARI, J. 2019. Incremental answer set programming with overgrounding. *Theory Pract. Log. Program. 19,* 5-6, 957–973.

DE KLEER, J. 1986. An assumption-based TMS. *Artif. Intell. 28,* 2, 127–162.

DOYLE, J. 1979. A truth maintenance system. *Artif. Intell. 12,* 3, 231–272.

EITER, T., OGRIS, P., AND SCHEKOTIHIN, K. 2019. A distributed approach to LARS stream reasoning (system paper). *Theory Pract. Log. Program. 19,* 5-6, 974–989.

GAI, Y., KRISHNAMACHARI, B., AND JAIN, R. 2012. Combinatorial Network Optimization With Unknown Variables: Multi-Armed Bandits With Linear Rewards and Individual Observations. *IEEE/ACM Transactions on Networking 20,* 5, 1466–1478.

GASCHNIG, J. 1979. Performance measurement and analysis of certain search algorithms. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA.

GEBSER, M., GROTE, T., KAMINSKI, R., OBERMEIER, P., SABUNCU, O., AND SCHAUB, T. 2012. Stream reasoning with answer set programming: Preliminary report. In *KR*. AAAI Press.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2019. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program. 19,* 1, 27–82.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *ICLP/SLP*. MIT Press, 1070–1080.

GENT, I. P., JEFFERSON, C., AND NIGHTINGALE, P. 2017. Complexity of n-queens completion. *J. Artif. Intell. Res. 59*, 815–848.

GOMES, C. P., SELMAN, B., AND KAUTZ, H. A. 1998. Boosting combinatorial search through randomization. In *AAAI/IAAI*. AAAI Press / The MIT Press, 431–437.

HEHENBERGER, P., VOGEL-HEUSER, B., BRADLEY, D., EYNARD, B., TOMIYAMA, T., AND ACHICHE, S. 2016. Design, modelling, simulation and integration of cyber physical systems: Methods and applications. *Comput. Ind. 82*, 273–289.

HUANG, J. 2007. The effect of restarts on the efficiency of clause learning. In *IJCAI*. 2318–2323.

KAUFMANN, B., LEONE, N., PERRI, S., AND SCHAUB, T. 2016. Grounding and solving in answer set programming. *AI Magazine 37,* 3, 25–32.

NADEL, A. AND RYVCHIN, V. 2012. Efficient SAT solving under assumptions. In *SAT*. 242–255.

PIPATSRISAWAT, K. AND DARWICHE, A. 2007. A lightweight component caching scheme for satisfiability solvers. In *SAT*. 294–299.

RATASICH, D., KHALID, F., GEISSLER, F., GROSU, R., SHAFIQUE, M., AND BARTOCCI, E. 2019. A roadmap toward the resilient internet of things for cyber-physical systems. *IEEE Access 7*, 13260–13283.

ROSSI, D. AND ROSSINI, G. 2012. On sizing CCN content stores by exploiting topological information. In *INFOCOM Workshops*. IEEE, 280–285.

SILVA, J. P. M. AND SAKALLAH, K. A. 1996. Conflict analysis in search algorithms for satisfiability. In *ICTAI*. IEEE Computer Society, 467–469.

SUTTON, R. S. 1995. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *NIPS*. MIT Press, 1038–1044.

SUTTON, R. S. AND BARTO, A. G. 2018. *Reinforcement Learning: An Introduction*, 2nd ed.

SWIFT, T. AND WARREN, D. S. 2012. XSB: extending prolog with tabled logic programming. *Theory Pract. Log. Program. 12,* 1-2, 157–187.