Sketched Answer Set Programming

Sergey Paramonov, Christian Bessiere, Anton Dries, Luc de Raedt

Sketched Answer Set Programming

Sergey Paramonov
KU Leuven
Leuven, Belgium
sergey.paramonov@kuleuven.be

Christian Bessiere
LIRMM CNRS
Montpellier, France
bessiere@lirmm.fr

Anton Dries
KU Leuven
Leuven, Belgium
anton.dries@kuleuven.be

Luc De Raedt
KU Leuven
Leuven, Belgium
luc.deraedt@kuleuven.be

***Abstract*—Answer Set Programming (ASP) is a powerful modeling formalism for combinatorial problems. However, writing ASP models can be hard. We propose a novel method, called Sketched Answer Set Programming (SkASP), aimed at facilitating this. In SkASP, the user writes partial ASP programs, in which uncertain parts are left open and marked with question marks. In addition, the user provides a number of positive and negative examples of the desired program behaviour. SkASP then synthesises a complete ASP program. This is realized by rewriting the SkASP program into another ASP program, which can then be solved by traditional ASP solvers. We evaluate our approach on 21 well known puzzles and combinatorial problems inspired by Karps 21 NP-complete problems and on publicly available ASP encodings.**

***Index Terms*—inductive logic programming, constraint learning, answer set programming, sketching, constraint programming, relational learning**

I. Introduction

Many AI problems can be formulated as constraint satisfaction problems that can be solved by state-of-the-art constraint programming (CP) [34] or answer set programming (ASP) techniques [27]. Although these frameworks provide declarative representations that are in principle easy to understand, writing models in such languages is not always easy.

On the other hand, for traditional programming languages, there has been significant attention for techniques that are able to complete [25] or learn a program from examples [17]. The idea of program sketching is to start from a sketched program and some examples to complete the program. A sketched program is essentially a program where some of the tests and constructs are left open because the programmer might not know what exact instruction to use. For instance, when comparing two variables X and Y, the programmer might not know whether to use $X < Y$ or $X \leq Y$ or $X > Y$ and write X ?= Y instead (while also specifying the domain of ?=, that is, which concrete operators are allowed). By providing a few examples of desired program behaviour and a sketch, the target program can then be automatically found. Sketching is thus a form of "lazy" programming as one does not have to fill out all details in the programs; it can also be considered as program synthesis although there are strong syntactic restrictions on the programs that can be derived; and it can be useful for repairing programs once a bug in a program has been detected. Sketching has been used successfully in a number of applications [24], [35], [19] to synthesise imperative programs. It is these capabilities that this paper brings to the field of ASP.

This work has been partially funded by the ERC AdG SYNTH (Synthesising inductive data models)

As a motivating example assume one needs to solve the Peacefully Coexisting Armies of Queens, a version of the n-queens problem with black and white queens, where queens of the same color do not attack each other. One might come up with the following sketched program (where R_w (C_b) stand for the variable representing the row (column) of a white (black) queen):

Sketch 1: Peacefully Coexisting Armies of Queens

```
1  :- queen(w,Rw,Cw), queen(b,Rb,Cb), Rw ?= Rb.
2  :- queen(w,Rw,Cw), queen(b,Rb,Cb), Cw ?= Cb.
3  :- queen(w,Rw,Cw), queen(b,Rb,Cb), Rw ?+ Rb ?= Cw ?+ Cb.
```

This program might have been inspired by a solution written in the constraint programming language Essence available from the CSP library [32]. Intuitively, the sketched ASP specifies constraints on the relationship between two queens on the rows (first rule), columns (second rule) and diagonals (third rule), but it expresses also uncertainty about the particular operators that should be used between the variables through the built-in alternatives for ?= (which can be instantiated to one of $=, \neq, <, >, \leq, \geq$) and for ?+ (for arithmetic operations). When providing an adequate set of examples to the ASP, the SkASP solver will then produce the correct program.

The key contributions of this paper are the following: 1) we adapt the notion of sketching for use with Answer Set Programming; 2) we develop an approach (using ASP itself) for computing solutions to a sketched Answer Set Program; 3) we contribute some simple complexity results on sketched ASP; and 4) we investigate the effectiveness and limitations of sketched ASP on a dataset of 21 typical ASP programs.

II. ASP and Sketching

Answer Set Programming (ASP) is a form of declarative programming based on the stable model semantics [15] of logic programming [27]. We follow the standard syntax and semantics of ASP as described in the Potassco project [13]. A *program* is a set of rules of the form $a \leftarrow a_1, \ldots a_k, not\ a_{k+1}, \ldots, not\ a_n$ A positive or negative atom is called a *literal*, a is a positive propositional literal, called a *head*, and for i between 1 and k, a_i is a positive propositional atom; and for i between $k+1$ and n, $not\ a_i$ is a negative propositional literal. The *body* is the conjunction

```
[SKETCH]
reached(Y) :- cycle(a,Y).
reached(Y) :- cycle(X,Y), reached(X).
    :- ?p(Y), ?not ?q(Y).
[EXAMPLES]
positive: cycle(a,b). cycle(b,c). ...
negative: cycle(a,b). cycle(b,a).
[SKETCHEDVAR]
?p/1 : node, reached
?q/1 : node, reached
[FACTS]
node(a). node(b). node(c).
[EXAMPLES]
positive: cycle(a,b). cycle(b,c)...
negative: cycle(a,b). cycle(b,a).
```

(a) Hamiltonian Cycle (ASP due to [13])

```
%%%%% EXAMPLES AND DECISIONS %%%%%%
positive(0). cycle(0,a,b). cycle(0,b,c). cycle(0,c,a).
reified_q_choice(c_node). reified_q_choice(c_reached).
1 {decision_q(X) : reified_q_choice(X)} 1.
1 {decision_not(pos) ; decision_not(neg)} 1.
%%%%% INFERENCE RULES %%%%%%
reified_q(E,c_node,X0) :- node(X0),examples(E).
reified_q(E,c_reached,X0) :- reached(E,X0).
reached(E,Y) :- cycle(E,a,Y), examples(E).
reached(E,Y) :- cycle(E,X,Y), reached(E,X), examples(E).
reified_not(E,pos,Q,Y) :- reified_q(E,Q,Y).
reified_not(E,neg,Q,Y) :- not reified_q(E,Q,Y), dom1(Y),dom2(Q),examples(E).
%%%%% POSITIVE/NEGATIVE SKETCHED RULES %%%%%%
:- reified_p(E,P,Y), reified_not(E,Not_D,Q,Y), decision_p(P), decision_q(Q),
                decision_not(Not_D), positive(E).
neg_sat(E) :- reified_p(E,P,Y),   reified_not(E,Not_D,Q,Y), decision_p(P),
decision_q(Q), decision_not(Not_D), negative(E).
:- not neg_sat(E), negative(E).
```

(b) ASP core of rewritten Fig. 1a (only predicate q shown, domains, facts, etc omitted)

```
[SKETCH] %constraints on squares, rows, columns
:- cell(X,Y,N), cell(X,Z,M), Y ?= Z, N ?= M.
:- cell(X,Y,N), cell(Z,Y,M), X ?= Z, N ?= M.
insquare(S,N) :- cell(X,Y,N), square(S,X,Y).
:- num(N), squares(S), ?not insquare(S,N).
```

(c) Sudoku (core). ASP code from [18]

```
[SKETCH] %constraints on rows and columns
:- cell(X,Y,N), cell(X,Z,M), Y ?= Z, N ?= M.
:- cell(X,Y,N), cell(Z,Y,N), X ?= X, N ?= M.
[EXAMPLES]
positive: cell(1,1,a). cell(1,2,b). cell(1,3,c)...
```

(d) Latin Square (based on Sudoku; core)

Fig. 1: Collection of sketches and an example of rewriting used in the paper

of the literals. A rule of the form $a \leftarrow .$ is called a *fact* and abbreviated as $a.$ and a rule without a head specified is called an *integrity constraint* (a is $\bot$ in this case). *Conditional literals*, written as $a : l_1, \ldots, l_n$, and *cardinality constraints*, written as $c_{\min}\{l_1, \ldots, l_n\}c_{\max}$, are used ($l_1, \ldots, l_n$ are literals here, and $c_{\min}, c_{\max}$ are non-negative integers). A conditional atom holds if its condition is satisfied and a cardinality constraint is satisfied if between $c_{\min}$ and $c_{\max}$ literals hold in it. Furthermore, as ASP is based on logic programming and also allows for *variables*, denoted in upper-case, the semantics of a rule or expression with a variable is the same as that of its set of ground instances. We restrict the ASP language to the NP-complete subset specified here. For more details on ASP, see [13], [10].

We extend the syntax of ASP with *sketched* language constructions. Instead of allowing only atoms of the form $p(t_1,...,t_n)$, where p/n is a predicate and the t_i are terms (variables or constants), we now allow to use *sketched atoms* of the form $?q(t_1,...,t_n)$ where $?q$ is a *sketched predicate variable* with an associated domain d_q containing actual predicates of arity n. The meaning of the sketched atom $?q(t_1,...,t_n)$ is that it can be replaced by any real atom $p(t_1,...,t_n)$ provided that $p/n \in d_q$. It reflects the fact that the programmer does not know which p/n from d_q should be used. Sketched atoms can be used in the same places as any other atom.

We also provide some syntactic sugar for some special cases and variants, in particular, we use a *sketched inequality* $X\ ?{=}\ Y$, a *sketched* arithmetic operator $X\ ?{+}\ Y$ (strictly speaking, this is not a sketched predicate but an operator, but we only make this distinction where needed), and *sketched negation* $?not\ p(X)$ (which is, in fact, a sketched operator of the form "?not ¡atom¿"; it always has as input a positive atom and its domain is {atom, -atom}, where -atom is a syntactically new atom, which represents the negation of the original atom). The domain of $X\ ?{=}\ Y$ is the set $\{=, \neq, <, >, \geq, \leq, \top\}$, where $\top$ is the atom that is always satisfied by its arguments, the domain of $X\ ?{+}\ Y$ is the set $\{+, -, \times, \div, dist\}$ where $dist(a,b)$ is defined as $|a-b|$, and the domain of $?not$ is $\{\emptyset, not\}$. An example of sketched inequality can be seen in Line 2 of Figure 1c, examples of sketched predicates and negation in Line 4 of Figure 1a and sketched arithmetic in Line 3 of Sketch 1.

A sketched variable is a sketched predicate, a sketched negation, a sketched inequality or a sketched arithmetic operator. The set of all sketched variables is referred to as S. Predicate p *directly positively (negatively) depends* on q iff q occurs positively (negatively) in the body of a rule with p in the head or p is a sketched predicate and q is in its domain; p *depends (negatively) on* q iff (p,q) is in the transitive closure of the direct dependency relation. A sketch is *stratified* iff there is no negative cyclic dependency. We restrict programs to the stratified case. An *example* is a set of ground atoms.

A *preference* is a function from Θ (possible substitutions) to $\mathbb{Z}$. A substitution θ is *preferred* over θ' given preferences f iff for all $s_i \mapsto d_i \in \theta$ and $s_i \mapsto d_i' \in \theta'$ it holds that $f(s_i \mapsto d_i) \geq f(s_i \mapsto d_i')$ and at least one inequality is strict. First, when $f(\theta)$ is constant, all substitutions are equal and there are no preferences (all equally preferred). Because specifying preferences might impose an extra burden on the user, we also provide default preferences for the built-in sketched variables (like inequality, etc), cf. the experimental section.

The Language of Sketched Answer Set Programming (SkASP) supports some of the language features of ASP. The language of SkASP has the following characteristics:

- it allows for a set of rules of the form $a \leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m.$;
- predicates (such as a predicate p/n or comparison $\leq$) and operators (such as arithmethic $+, -, \times$, etc) in these rules can be sketched;

- aggregates can be used in the body of the rules as well (stratified; see Extension Section IV);
- the SkASP program has to be stratified;
- the choice rules are not allowed.

The key idea behind our method is that the SkASP program is rewritten into a normal ASP program (with choice rules, etc.) in order to obtain a solution through the use of an ASP solver. As we will see in Theorem 2: the language of SkASP stays within the complexity bounds of normal ASP, which makes the rewriting possible (SkASP$\mapsto$ASP).

Let us now formally introduce the problem of SkASP.

Definition 1 (The Problem of Sketched Answer Set Programming)**.** Given a sketched answer set program P with sketched variables S of domain D and preferences f, and positive and negative sets of examples $\mathbf{E}^+$ and $\mathbf{E}^-$, the *Sketched Answer Set Problem* is to find all substitutions $\theta : S \mapsto D$ preferred by f such that $P\theta \cup \{e\}$ has an answer for all e in $\mathbf{E}^+$ and for no e in $\mathbf{E}^-$. The decision version of SkASP asks whether there exists such a substitution θ.

III. Rewriting Schema

One might consider a baseline approach that would enumerate all instances of the ASP sketch, and in this way produce one ASP program for each assignment that could then be tested on the examples. This naive grounding and testing approach is, however, infeasible: the number of possible combinations grows exponentially with the number of sketched variables. E.g., for the sketch of the Radio Frequency Problem [7] there are around 10^5 possible assignments to the sketched variables. Multiplied by the number of examples, around a million ASP programs would have to be generated and tested. This is infeasible in practice.

The key idea behind our approach is to rewrite a SkASP problem $(P,S,D,f,\mathbf{E}^+,\mathbf{E}^-)$ into an ASP program such that the original sketching program has a solution iff the ASP program has an answer set. This is achieved by 1) inserting *decision variables* into the sketched predicates, and 2) introducing *example identifiers* in the predicates.

The original SkASP problem is then turned into an ASP problem on these decision variables and solutions to the ASP problem allow to reconstruct the SkASP substitution.

The rewriting procedure has four major steps: *example expansion*, *substitution generation*, *predicate reification* and *constraint splitting*. (Here we follow the notation on meta-ASP already used in the literature [21], [11].)

Example Identifiers To allow the use of multiple examples in the program, every relevant predicate is extended with an extra argument that represents the example identifier. The following steps are used to accommodate this in the program, denoted as *metaE(P,S,$\mathbf{E}^+$,$\mathbf{E}^-$)*.

1) Let *SP* be the set of all predicates that depend on a predicate occurring in one of the examples.
2) Replace each literal $p(t_1,...,t_n)$ for a predicate $p \in SP$ in the program P by the literal $p(E,t_1,...,t_n)$, where E is a variable not occurring in the program.
3) Add the guard $examples(E)$ (the index of all pos./neg. examples) to the body of each rule in P.
4) For each atom $p(t_1,...,t_n)$ in the i-th example, add the fact $p(i,t_1,...,t_n)$ to P.
5) For each positive example i, add the fact $positive(i)$ to P, and for each negative one, the fact $negative(i)$.

E.g., the rule in Line 2 of Figure 1a becomes Line 9 of Figure 1b, and the example in Line 14 is rewritten as in Line 2.

Substitution Generation We now introduce the decision variables, referred as *metaD(S,D)*:

1) For each sketched variable s_i with domain D_i

$$1\ \{decision\_s_i(X) : d_i(X)\}\ 1.$$

2) For each value v in D_i, add the fact $d_i(v)$.

This constraint ensures that each answer set has exactly one value from the domain assigned to each sketched variable. This results in a one-to-one mapping between decision atoms and sketching substitution θ. An example can be seen in Lines 4 and 5 of Figure 1b.

Predicate Reification We now introduce the reified predicates, referred as *metaR(S,D)*

1) Replace each occurrence of a sketched atom $s(t_1,...,t_n)$ in a rule of P with the atom $reified\_s(D,t_1,\ldots,t_n)$, and add $decision\_s(D)$ to the body of the rule.
2) For each sketched variable s and value d_i in its domain, add the following rule to P:

$$reified\_s(d_i,X_1,\ldots,X_n) \leftarrow d_i(X_1,\ldots,X_n).$$

where the first argument is the decision variable for s.

Thus, semantically $reified\_s(d_i,X_1,\ldots,X_n)$ is equivalent to $d_i(X_1,\ldots,X_n)$ and $decision\_s(d_i)$ indicates that the predicate d_i has been selected for the sketched variable s. Notice that we focused here on the general case of a sketched predicate $?p(\ldots)$. It is straightforward to adapt this for the sketched inequality, negation and arithmetic. Examples of reification can be seen in Lines 7 of Figure 1b for the sketched $?q$ of the sketch in Figure 1a and in Lines 11, 12 for reified negation.

Integrity Constraint Splitting (referred as *metaC(P)*)

1) Replace each integrity constraint $\leftarrow body$ by:

$$\leftarrow body, positive(E)$$

$$negsat(E) \leftarrow body, negative(E)$$

2) And add the rule to the program:

$$\leftarrow negative(E), not\ negsat(E).$$

This will ensure that all positives and none of the negatives have a solution. For example, the constraint in Line 4 of Figure 1a is rewritten into a positive constraint in Lines 14,15 and into a negative in Lines 16, 16, 17.

Another important result is that the preferences do not affect the decision complexity. Proofs can be found in the supplementary materials.

Theorem 1 (Sound and Complete Sketched Rewriting)**.** A sketched ASP program $(P,S,D,f,\mathbf{E}^+,\mathbf{E}^-)$ has a satisfying substitution θ iff the meta program $T = metaE(P,S,\mathbf{E}^+,\mathbf{E}^-) \cup metaD(S,D) \cup metaR(S,D) \cup metaC(P)$ has an answer set.

Interestingly, the sketched ASP problem is in the same complexity class as the original ASP program.

Theorem 2 (Complexity of Sketched Answer Set Problem)**.** The decision version of propositional SkASP is in NP.

Proof. Follows from the encoding of SkASP into a fragment of ASP which is in NP. □

Dealing with preferences Preferences are, as we shall show in our experiments, useful to restrict the number of solutions. We have implemented preferences using a post-processing approach (which will also allow to apply the schema to other formalisms such as CP or IDP [8]). We first generate the set of all solutions O (without taking into account the preferences), and then post-process O. Basically, we filter out from O any solution that is not preferred (using tests on pairs (o,o') from $O \times O$). The preferences introduce a partial order on the solutions. For example, assume $?p$ ($?q$) can take value p_1 (q_1) with preference of 1 and p_2 (q_2) with 2. If (p_1,q_2) and (p_2,q_1) are the only solutions, they are kept because they are incomparable – $(1,2)$ is not dominated by $(2,1)$ (and vice versa). If (p_1,q_1) is also solution, (p_1,q_2) and (p_2,q_1) are removed because they are dominated by (p_1,q_1).

While the number of potential Answer Sets is in general exponential for a sketched ASP, the number of programs actually satisfying the examples is typically rather small (in our experiments, below 10000-20000). If that is not the case, then the problem is under-constrained and it needs more examples. No user would be able to go over a million of proposed programs.

IV. System Extension: Aggregates and Use-Case

An aggregate *#agg* is a function from a set of tuples to an integer. For example,$\#count\{Column,Row : queen(Column,Row)\}$ counts the number of instances *queen(Column,Row)* at the tuple level. Aggregates are often useful for modeling. However, adding aggregates to non-disjunctive ASP raises the complexity of an AS existence check, unless aggregate dependencies are stratified [12]. It is possible to add aggregates into our system under the following restrictions: stratified case, aggregates occur in the body in the form $N = \#agg\{\ldots\}$, sketched with the keyword `?#` , where *#agg* can be *max*, *min*, *count* and *sum*. This immediately allows us to model problems such as Equal Subset Sum (for details, see the repository), where one needs to split a list of values, specified as a binary predicate `val(ID,Value)` into two subsets, such as `subset1(ID)` (and `subset2(ID)` respectively), such that the sum of both subsets is equal. Essentially, we sketch the constraint of the form:

```
:- S1 != S2, S1 = ?#{V,X:val(X,V),subset1(X)}...
```

Formally, each aggregate can be seen as an expression of the form:

$$S = \#agg\{Z_1,\ldots,Z_n : cond(\overbrace{X_1,\ldots,X_k}^{internal},\overbrace{Y_1,\ldots,Y_h}^{external},\overbrace{Z_1,\ldots,Z_n}^{aggregated})\}, external(Y_1,\ldots,Y_h)$$

where S is an integer output, and $Y_1,\ldots,Y_h$, shortened as $\bar{Y}$ ($\bar{X}$ and $\bar{Z}$ are the same kind of shortening) are bound to other atoms in the rule, to which we refer as *external($\bar{Y}$)* ("external" with respect to the condition in the aggregate; it is simply shortening for a conjunction of atoms, which share variables with the condition in the predicate).

To give an example of $\bar{X},\bar{Y},\bar{Z}$ in a simple context: if we were to compute an average salary per department in a company, we might have written a rule of the form:

```
avg_sal(A,D) :- A = #avg{S,N: salaries(N,S,D)},
                                  department(D).
```

Then, $\bar{Z}$ consists of the variable `S` and `D` is the external variable (with respect to the condition in the aggregate), i.e., $\bar{Y}$ and $\bar{X}$ is composed out of the variable `N`, since it is neither used in the aggregation, nor in the other atoms outside of the aggregate.

A sketched aggregate ?#, can be reified similarly to the regular sketched atoms, i.e.:

$$reified(S,sum,\bar{Y}) \leftarrow S = \#sum\{\bar{Z} : cond(\bar{X},\bar{Y},\bar{Z})\}, external(\bar{Y}).$$

similarly for other aggregate functions; the same rules, e.g., the example extension, apply to aggregate reification.

With aggregates we can sketch a significantly larger class of problems. Consider the problem from the Functional Pearls Collection: "Finding celebrities problem" [5][1]. Problem statement: "Given a list of people at a party and for each person the list of people they know at the party, we want to find the celebrities at the party. A celebrity is a person that everybody at the party knows but that only knows other celebrities. At least one celebrity is present at the party." The sketch core looks as follows (names are shortened):

```
n(N) :- N = ?#{ P : p(P) }.
:- c(C), p(C), n(N), S = ?#{P : k(P,C), p(P)},
                                  S < N-1.
:- c(C), p(C), not c(P), k(C,P).
```

The last rule is an integrity constraint verifying that no celebrity, `c`, knows a person who is not a celebrity. The first line sketches a rule that should find what aggregation metric on the people (unary predicate person, `p`) should be used in the problem. The sketched rule in the second line makes use of this metric, denoted as `n`, and says that an aggregation should be performed on the binary "knows" predicate, `k`, (indicating that two persons know each other); so the outcome of the sketched aggregation on the connection between people should be compared to an overall metric on all people individually.

V. Experimental Evaluation

For the experimental evaluation we have created a dataset consisting of 21 classical combinatorial problems among

[1] ASP code: hakank.org/answer_set_programming/finding_celebrities.lp4

Problem	# Sketched	# ?=	# ?+	# ?not	# ?p	# Rules
Graph Clique	3	1	0	0	2	4
3D Matching	3	3	0	0	0	1
Graph Coloring	7	4	0	0	3	2
Domination Set	3	0	0	1	2	5
Exact Cover	7	2	0	1	4	3
Sudoku	5	4	0	1	0	4
B&W Queens	5	3	2	0	0	3
Hitting Set	3	0	0	1	2	2
FAP	3	0	0	1	2	3
Feedback Arc Set	4	0	0	2	2	3
Latin Square	4	4	0	0	0	2
Edge Domination	3	0	0	1	2	5
FAP	5	3	2	0	0	3
Set Packing	4	2	0	0	2	1
Clique Cover	4	3	0	1	0	3
Feedback Set	5	0	0	5	0	3
Edge Coloring	3	3	0	0	0	3
Set Splitting	5	2	0	1	2	3
N Queens	6	4	2	0	0	3
Vertex Cover	3	0	0	1	2	4
Subg. Isomorph.	5	2	0	1	2	4

TABLE I: Dataset summary: the number of sketched variables, of rules, of particular types of sketched variables, e.g., "# ?not", indicates how many atoms with the sketched negation are in the program.

which most are NP-complete. For the problem list and precise sketch specifications used in the experiments, we refer to Table I. All problems, their code, and implementation details, can be found in the accompanying Github repository: https://github.com/SergeyParamonov/sketching

Dataset of Sketches. The key challenge in evaluating program synthesis techniques such as SkASP is the absence of benchmark datasets (as available in more typical machine learning tasks). At the same time, although there are many example ASP programs available in blogs, books or come with software, these typically employ advanced features (such as incremental grounding, optimization or external sources) which are not supported by SkASP as yet. Therefore we had to design our own dataset in a systematic way (and put it in the public domain). The dataset is based on a systematic concept (the 21 problems by Karp). When we could find encodings for these problem (such as Sudoku in Figure 1c from [18] and Hamiltonian Cycle in Figure 1a from [13]) we took these problems, in all other cases we developed a solution according to the standard generate and test development methodology of ASP. Specifically (see $\mathbf{Q}_5$) we looked for different encodings in the public domain of ASPs favorite – the N-queens problem (these encoding can tackle even its NP-complete version [16]).

After creating all the ASP programs, we turned them into sketches by looking for meaningful opportunities to use sketched variables. We introduced sketched variables to replace operators (equalities and inequalities), to replace arithmetic (such as plus and minus) and to decide whether to use negated literals or not, and to make abstraction of predicates for which another predicate existed with the same signature.

Finally, we had to select the examples in a meaningful way, that is, we selected examples that would be informative (as a user of SkASP would also do). Positive examples were actually selected more or less random, negative examples are meant to violate one or more of the properties of the problem. Furthermore, we also tried to select examples that carry different information (again as a user of SkASP would do). We selected between 4 and 7 examples for each model. Where relevant in the experiments, we sampled the sketched variables (e.g. $\mathbf{Q}_5$) or the examples (e.g. $\mathbf{Q}_3$)

Experimental questions are designed to evaluate how usable is SkASP in practice. Users want in general to provide only a few examples ($\mathbf{Q}_1$-$\mathbf{Q}_3$), to obtain a small number of solutions (ideally only one) ($\mathbf{Q}_1$-$\mathbf{Q}_2$), the examples should be small ($\mathbf{Q}_4$), the solutions should be correct (all), want to know whether and when to use preferences ($\mathbf{Q}_2$), and how robust the technique is to changes in the encoding ($\mathbf{Q}_5$) as it is well known in ASP that small changes in the encoding can have large effects. Finally, they are typically interested in how the learned programs change as the sketches become more complex ($\mathbf{Q}_3$). With this in mind, we have designed and investigated the following experimental questions:

- $\mathbf{Q}_1$: What is the relationship between the number of examples and the number of solutions? How many examples does it take to converge?
- $\mathbf{Q}_2$: Are preferences useful?
- $\mathbf{Q}_3$: What is the effect of the number of sketched variables on the convergence and correctness of the learned programs?
- $\mathbf{Q}_4$: Do models learned on examples with small parameter values generalize to models with larger parameter values?
- $\mathbf{Q}_5$: What is the effect of encoding variations on the number of solutions and their correctness?

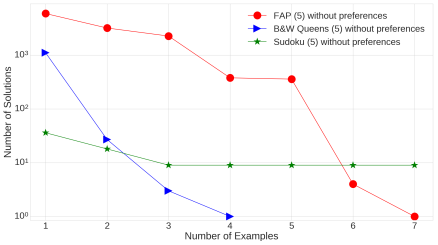
Implementation details and limitations. The SkASP engine is written in Python 3.4 and requires pyasp. All examples have been run on a 64-bit Ubuntu 14.04, tested in Clingo 5.2.0. The current implementation does not support certain language constructs such as choice rules or optimization.

We use the *default preferences* in the experiments for the built-in inequality sketch X ?= Y: namely $=$ and $\neq$ have equal maximal preference. A user can redefine the preferences. Our experiments indicate that for other sketched types (e.g., arithmetic, etc) no default preferences are needed.

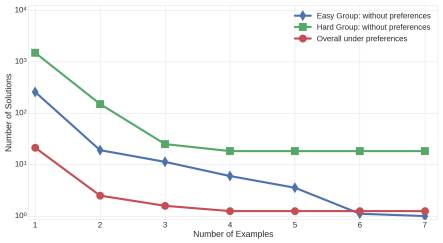
We investigate $\mathbf{Q}_1$ by measuring the impact of the number of examples on the number of solutions of the 21 SkASP problems. An interesting observation is that even if the user wants to solve, say the Latin Square 20×20, she does not need to provide examples of size $20 \cdot 20 = 400$. She can simply provide 3×3 examples and our SkASP problem will learn the generic Latin Square program (see Figure 1d).

Figure 2a shows how the number of solutions for some of our 21 SkASP problems depends on the number of examples. In some cases, 7 examples are sufficient to converge to a single solution e.g., FAP, B&W Queens.

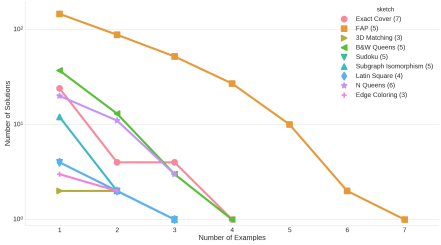
On some other problems, however, after 7 examples there still remain many solutions (on average 18 for problems that do not converge). Figure 2b reports the same information as Figure 2a for all 21 problems: the average number of solutions; the average on the 9 that converge within 7 examples, referred to as the *easy problems*; and the average on the 12 that still have several solutions after 7 examples, referred to as the



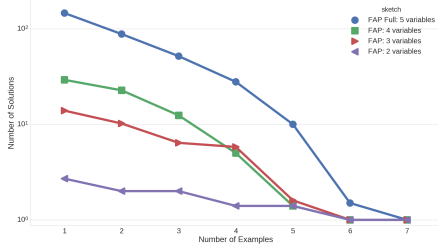
(a) Convergence without preferences (with 5 sketched variables): B&W Queens (Sketch 1) and FAP converge, while Sudoku does not



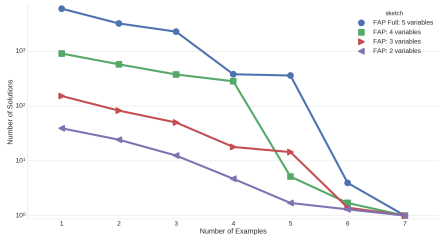
(b) Average number of solutions over the dataset, split into the *easy* group converging without preferences and *hard* not converging



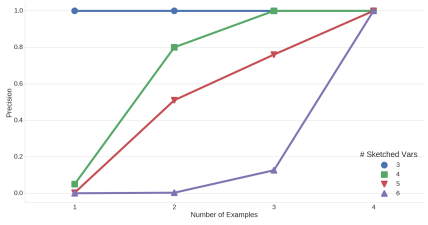
(c) Between 2 and 7 examples are needed to obtain a unique solution (or a small group of equivalent ones) under preferences.



(d) The effect of the number of sketched variables on the solutions with preferences

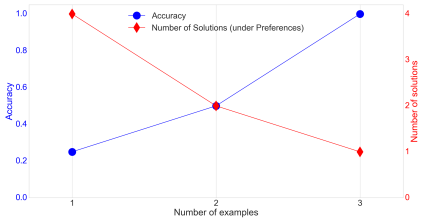


(e) The effect of the number of sketched variables on the solutions without preferences

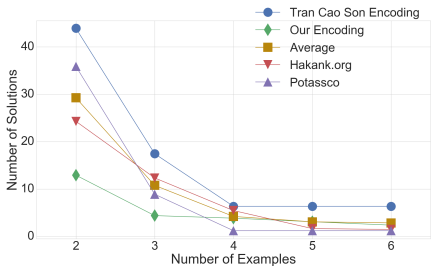


(f) Dependency between the number of sketched variables and precision

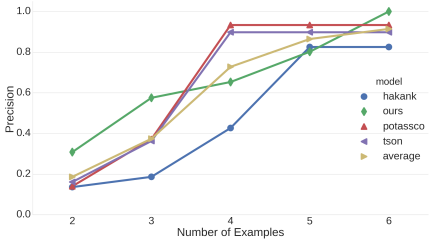
Fig. 2: Addressing experimental questions $\mathbf{Q}_1$-$\mathbf{Q}_3$. $\mathbf{Q}_1$ and $\mathbf{Q}_2$: convergence without preferences (a), across the dataset (b), for each problem (c). $\mathbf{Q}_3$: the effect of different number of sketched variable on the number of solutions and precision, FAP (d, e); *N*-queens (f). log-scale (a-e)



(a) Measuring the generalized accuracy and number of solutions. Latin Square 4×4.



(b) The number of solutions (default preferences) for various *N*-queen encodings



(c) Correctness for *N*-queens various encodings (precision)

Fig. 3: Addressing experimental questions $\mathbf{Q}_4$ and $\mathbf{Q}_5$: generalization accuracy on Latin Square (a), the effect of the encoding variations on the number of solutions (b), on precision (c).

hard problems. When SkASP does not converge to a unique solution, this leaves the user with choices, often amongst equivalent ASP programs, which is undesirable.

For problems that do not converge after a few examples, we propose to use preferences, as provided by our SkASP framework. We use the default preference described earlier.

We investigate $\mathbf{Q}_2$ by measuring again the impact of the number of examples on the number of solutions. In Figure 2c, we observe that all problems have converged in less than 7 examples (under default preferences). The impact of preferences on the speed of convergence is even more visible on the whole set of problems, as reported in Figure 2b. The number of solutions with preferences is smaller, and often much smaller than without preferences, whatever the number of examples provided. With preferences, all our 21 problems are learned with 7 examples.

To analyze the number of solutions in $\mathbf{Q}_3$, we look into the convergence of FAP by varying the number of sketched variables. The original sketched program of FAP contains 5 sketched variables. We vary it from 2 to 5 by turning 3, 2, 1, or 0 sketched variables into their actual value (chosen randomly and repeated over multiple runs). As expected, in Figure 2d, we observe that the more there are sketched variables in the SkASP, the slower the number of solutions decreases. Furthermore, the number of sketched variables has a greater impact on the convergence without preferences, as we see in Figure 2e. After 3-5 examples under preferences we have fewer than 10 solutions, while without preferences there are still dozens or hundreds of solutions.

To analyze *correctness* in $\mathbf{Q}_3$, we need first to define it. Informally, we mean that the program classifies arbitrary examples correctly, i.e., positive as positive, etc. A typical

metric to measure this is *accuracy*. However, there are no well defined arbitrary positive and negative examples for the most problems: what is an arbitrary random example for Feedback Arc Cover? Problems like Sudoku and N-queens do have standard examples because they are parameterized with a single parameter, which has a default value. Furthermore, for the standard 8-queens we know all solutions analytically, i.e., 92 combinations. Another issue is that the negative and positive classes are unbalanced. The usual way to address this issue is to use *precision*, i.e., $\frac{\text{True Positive}}{\text{True+False Positives}}$. (Recall is typically one because the incorrect programs produce way too many solutions that include the correct ones.) In Figure 2f, we see that in all cases we were able to reach the correct solution (here the locations of sketched variables were fixed as specified in the dataset); while increasing the number of sketched variables generally decreases the precision.

To investigate $\mathbf{Q}_4$, we have used the Latin Square from Listing 1d. We have used examples for Latin Square 3×3, and verified its correctness on Latin Square 4×4 (which can be checked analytically because all solutions are known). We have discovered, that there is an inverse dependency between number of solutions and accuracy, see Figure 3a. This happens because there are typically very few useful or "intended" programs while there are lot of incorrect ones.

To investigate $\mathbf{Q}_5$, we have focused on the N-queens problem and collected several encodings from multiple sources: Potascco, Hakank.org, an ASP course by Tran Cao Son[2] and our encoding. Whereas all the encodings model the same problem they show significant variations in expressing constraints. To reduce the bias in how the sketched variables are introduced and systematically measure the parameters, we pick sketched variables randomly (inequalities and arithmetic) and use the same examples from our dataset (randomly picking the correct amount) for all models.

In Figure 3b, while there is a certain variation in the number of solutions, they demonstrate similar behavior. For each encoding we have introduced 5 sketched variables and measured the number of solutions and precision. In Figure 3c we see that there is indeed a slight variation in precision, with 3 out of 4 clearly reaching above 90% precision, one reaching 100% and one getting 82%. Thus, despite variations in encoding, they generaly behave similarly on the key metrics. The results have been averaged over 100 runs.

Overall, we observe that only few examples are needed to converge to a unique or a small group of equivalent solutions. An example where such equivalent solutions are found is the edge coloring problem, where two equivalent (for undirected graphs) constraints are found:

$$\leftarrow color(X,Y_1,C), color(X,Y_2,C),\ Y_1 \neq Y_2.$$
$$\leftarrow color(X_1,Y,C), color(X_2,Y,C),\ X_1 \neq X_2.$$

For this problem these two constraints are equivalent and cannot be differentiated by any valid example.

[2] www.hakank.org/answer_set_programming/nqueens.lp
www.cs.uni-potsdam.de/~torsten/Lehre/ASP/Folien/asp-handout.pdf
www.cs.nmsu.edu/~tson/tutorials/asp-tutorial.pdf

An interesting observation we made on these experiments is that the hardness (e.g., in terms of runtime) of searching for a solution of a problem is not directly connected to the hardness of learning the constraints of this problem. This can be explained by the incomparability of the search spaces. SkASP searches through the search space of sketched variables, which is usually much smaller than the search space of the set of decision variables of the problem to learn.

VI. Related Work

The problem of sketched ASP is related to a number of topics. First, the idea of sketching originates from the area of programming languages, where it relates to so called self-completing programs [25], typically in C [24] and in Java [19], where an imperative program has a question mark instead of a constant and a programmer provides a number of examples to find the right substitution for it. While sketching has been used in imperative programming languages, it has – to the best of the authors' knowledge – never been applied to ASP and constraint programming. What is also new is that the sketched ASP is solved using a standard ASP solver, i.e., ASP itself.

Second, there is a connection to the field of inductive (logic) programming (ILP) [9], [28], [17]. An example is meta-interpretive learning [29], [30] where a Prolog program is learned based on a set of higher-order rules, which act as a kind of template that can be used to complete the program. However, meta-interpretive learning differs from SkASP in that it induces full programs and pursues as other ILP methods a search- and trace-based approach guided by generality, whereas SkASP using a constraint solver (i.e., ASP itself) directly. Furthermore, the target is different in that ASPs are learned, which include constraints. SkASP relates to meta-interpretation in ASP [11] in rule and decision materialization. The purpose is, however, different: they aim at synthesizing a program of higher complexity (Σ_2^P) given programs of lower complexity (*NP* and *Co-NP*).

There are also interesting works in the intersection of ILP, program synthesis and ASP [21], [23], [33]. The ILASP system [22] learns an ASP program from examples, and a set of modes, while minimizing a metric, typically the number of atoms. This program, learned completely from scratch, is not necessarily the best program from the user's point of view and may limit the possibility to localize the uncertainty based on the user's knowledge of the problem. Indeed, if all sketched predicates are added in the modes with corresponding background knowledge, then the set of hypotheses of sketched ASP is a subset of ILASP. However, if we specify a sketched constraint `:- p(X),q(Y),X?=Y` with the negative example {`p(1),q(2)`} as modes for ILASP [22], it would learn a program like `:- p(X)` (minimal program), but that is clearly not the program intended by the sketch. Furthermore, we compute all preferred programs instead of a single solution.

Third, there is also work on constraint learning, where the systems such as CONACQ [4], [2] and QUACQ [3] learn a set of propositional constraints, and ModelSeeker [1] learns global constraints governing a particular set of examples. The

subject has also been investigated in ILP setting [20]. However, the idea in all these approaches is to learn the complete specification of CSPs from scratch. Our setting is probably more realistic from a user perspective as it allows to use the knowledge that the user no doubt possesses about the underlying problem, and also requires much less examples. On the other hand, SkASP also makes, as other sketching approaches, the strong assumption that the intended target program is an instance of the sketched one. This may not always be true, for instance, when rules are missing in the program. This is an interesting issue for further research.

Fourth, our approach is related to debugging of ASP [14], [31]. Unlike SkASP such debuggers can be used to locate bugs, but typically do not provide help in fixing them. On the other hand, once a bug is identified, SkASP could be used to repair it by introducing a sketch and a number of examples[3] The approach of [26] is based on classical ILP techniques of generalization and specification and does not provide the freedom to indicate uncertain parts of the program.

VII. Discussion and Conclusions

Our contribution is four-fold: we have introduced the problem of sketched ASP; we have provided a rewriting schema for SkASP; we have created a dataset of sketches and we have evaluated our approach empirically demonstrating its efficiency and effectiveness.

User interaction is an interesting future direction, namely to suggest constraints and examples. For the former, if we are not able to reject a negative example, we can construct a constraint that would reject the negative examples and none of the positive examples. As for the examples, if we have two solutions to a problem, we can generate an example discriminating between them and ask user to clarify it, while this might not always be possible, since symmetric assignments might lead to semantically identical programs. In practice, however, this might be an important addition to simplify sketching for end users. Another direction is to incorporate non-constant preference handling into the model using the extensions of ASP for preference handling, such as *asprin* [6].

References

[1] Beldiceanu, N., Simonis, H.: A model seeker: Extracting global constraint models from positive examples. In: CP. pp. 141–157 (2012)

[2] Bessiere, C., Koriche, F., Lazaar, N., O'Sullivan, B.: Constraint acquisition. Artif. Intell. In Press (2017)

[3] Bessiere, C., Coletta, R., Hebrard, E., Katsirelos, G., Lazaar, N., Narodytska, N., Quimper, C., Walsh, T.: Constraint acquisition via partial queries. In: IJCAI. pp. 475–481 (2013)

[4] Bessiere, C., Coletta, R., Koriche, F.: A sat-based version space algorithm for acquiring constraint satisfaction problems. In: ECML. pp. 23–34. Springer (2005)

[3] During the experiments, we stumbled upon a peculiar bug. One ASP encoding that we discovered in a public repository worked mostly by pure luck. The following constraint `:-queen(X1,Y1),queen(X2,Y2),X1<X2,abs(Y1-X1)==abs(Y2-X2).` works because `abs` is not actually absolute value but an uninterpreted function, essentially it checks $X == Y$, and that is indeed the found solution. (This kind of bugs would be extremely hard to find using traditional debuggers, since technically the encoding produced correct solutions.). Also, while working on the aggregate extension use-case, we discovered a subtle bug: the case of a single celebrity was not handled correctly. In both cases, the author has been contacted and models have been updated.

[5] Bird, R., Curtis, S.: Functional pearls: Finding celebrities: A lesson in functional programming. J. Funct. Program. 16(1), 13–20 (Jan 2006)

[6] Brewka, G., Delgrande, J., Romero, J., Schaub, T.: Asprin: Customizing answer set preferences without a headache. In: AAAI. pp. 1467–1474 (2015)

[7] Cabon, B., de Givry, S., Lobjois, L., Schiex, T., Warners, J.: Radio link frequency assignment. Constraints 4(1), 79–89 (Feb 1999)

[8] de Cat, B., Bogaerts, B., Bruynooghe, M., Denecker, M.: Predicate logic as a modelling language: The IDP system. CoRR abs/1401.6312 (2014)

[9] De Raedt, L.: Logical and Relational Learning: From ILP to MRDM (Cognitive Technologies). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2008)

[10] Eiter, T., Ianni, G., Krennwallner, T.: Reasoning web. semantic technologies for information systems. chap. Answer Set Programming: A Primer, pp. 40–110 (2009)

[11] Eiter, T., Polleres, A.: Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. TPLP 6(1-2), 23–60 (2006)

[12] Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming. Artificial Intelligence 175(1), 278 – 298 (2011), john McCarthy's Legacy

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

[14] Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A meta-programming technique for debugging answer-set programs. In: AAAI

[15] Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP. vol. 88, pp. 1070–1080 (1988)

[16] Gent, I.P., Jefferson, C., Nightingale, P.: Complexity of n-queens completion. J. Artif. Intell. Res. 59, 815–848 (2017)

[17] Gulwani, S., Hernandez-Orallo, J., Kitzelmann, E., Muggleton, S.H., Schmid, U., Zorn, B.: Inductive programming meets the real world. Commun. ACM 58(11), 90–99 (2015)

[18] Hölldobler, S., Schweizer, L.: Answer set programming and clasp a tutorial. In: YSIP. p. 77 (2014)

[19] Jeon, J., Qiu, X., Foster, J.S., Lezama, A.S.: Jsketch: sketching for java. In: ESEC/FSE. pp. 934–937 (2015)

[20] Lallouet, A., Lopez, M., Martin, L., Vrain, C.: On learning constraint problems. In: ICTAI. pp. 45–52 (2010)

[21] Law, M., Russo, A., Broda, K.: Inductive Learning of Answer Set Programs, pp. 311–325. Springer International Publishing, Cham (2014)

[22] Law, M., Russo, A., Broda, K.: The ILASP system for learning answer set programs. https://www.doc.ic.ac.uk/~ml1909/ILASP (2015)

[23] Law, M., Russo, A., Broda, K.: Iterative learning of answer set programs from context dependent examples. TPLP 16(5-6), 834–848 (2016), https://doi.org/10.1017/S1471068416000351

[24] Lezama, A.S.: The sketching approach to program synthesis. In: APLAS. pp. 4–13 (2009)

[25] Lezama, A.S.: Program sketching. STTT 15, 475–495 (2013)

[26] Li, T., Vos, M.D., Padget, J., Satoh, K., Balke, T.: Debugging ASP using ILP. In: Technical Communications ICLP Cork, Ireland (2015)

[27] Lifschitz, V.: What is answer set programming? AAAI (2008)

[28] Muggleton, S., De Raedt, L.: Inductive logic programming: Theory and methods. Journal of Logic Programming 19(20), 629–679 (1994)

[29] Muggleton, S.H., Lin, D., Pahlavi, N., Tamaddoni-Nezhad, A.: Meta-interpretive learning: application to grammatical inference. Machine Learning 94(1), 25–49 (2014)

[30] Muggleton, S.H., Lin, D., Tamaddoni-Nezhad, A.: Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. Machine Learning 100(1) (2015)

[31] Oetsch, J., Pührer, J., Tompits, H.: Stepping through an Answer-Set Program, pp. 134–147. Springer Berlin Heidelberg (2011)

[32] Özgür, A.: CSPLib problem 110: Peaceably co-existing armies of queens. http://www.csplib.org/Problems/prob110 (2015)

[33] Ray, O.: Nonmonotonic abductive inductive learning. Journal of Applied Logic 7(3), 329 – 340 (2009), special Issue: Abduction and Induction in Artificial Intelligence

[34] Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Foundations of Artificial Intelligence, Elsevier (2006)

[35] Singh, R., Singh, R., Xu, Z., Krosnick, R., Solar-Lezama, A.: Modular synthesis of sketches using models. In: Verification, Model Checking, and Abstract Interpretation San Diego, USA. pp. 395–414 (2014)