Sketched Answer Set Programming: Extra Materials

1 Dataset Description

In Table 1, we provide an overview of the dataset and its properties, such the number of sketched variable in each problem and the number of rules, other columns denote the number of particular type of sketched variables, e.g., "# ?not" indicates how many atom with the sketched negation are in the program. Of course, one can introduce a different set of sketched variables for each problem but to give a more complete picture of the experimental settings we describe the dataset as it has been used in the experiments (where the number and position of sketched variables are fixed).

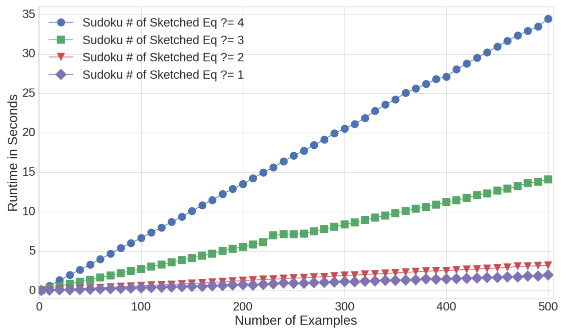
Problem	**# Sketched**	**# ?=**	**# ?+**	**# ?not**	**# ?p**	**# Rules**
Graph Clique	3	1	0	0	2	4
3D Matching	3	3	0	0	0	1
Graph Coloring	7	4	0	0	3	2
Domination Set	3	0	0	1	2	5
Exact Cover	7	2	0	1	4	3
Sudoku	5	4	0	1	0	4
B&W Queens	5	3	2	0	0	3
Hitting Set	3	0	0	1	2	2
FAP	3	0	0	1	2	3
Feedback Arc Set	4	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
Subgraph Isom-ism	5	2	0	1	2	4

Table 1: Dataset summary

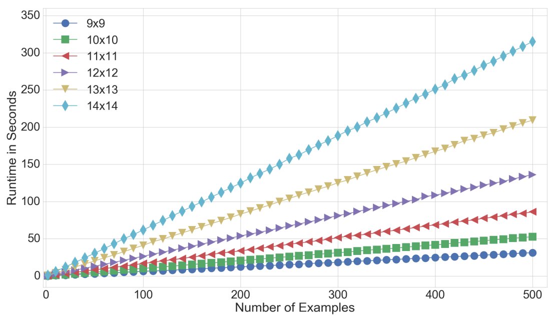
2 Runtime Experiments

Experimental questions:

- **Q**$_1$: What is the effect of the number of sketched variables on the runtime?
- **Q**$_2$: What is the effect of the example size (in atoms) on the runtime?
- **Q**$_3$: What is the runtime distribution between grounding and solving steps?
- **Q**$_4$: What is the effect of the encoding variation on the runtime?



(a) Sudoku: investigating the effect of different number of sketched variables on the runtime



(b) Latin Square: the effect of the example size on the runtime (the example size is quadratic in n)

Figure 1: Investigating $\mathbf{Q}_1$ and $\mathbf{Q}_2$: the effect of the number of sketched variables (left) and of the example size (right) on runtime

To address $\mathbf{Q}_1$ and $\mathbf{Q}_2$, we set up the following experiment: we have taken the 9×9 Sudoku encoding and varied the number of sketched inequalities in it. We then run it against randomly generated positive and negative examples (in equal quantities). The outcome is depicted in Figure 1a. We observe that the runtime depends on the number of sketched variables, i.e. the runtime exhibits a linear behavior with the slope determined by the number of sketched variables. To measure the effect of the examples size, i.e. the number of ground facts in a single example, we set up the following experiment: for the $n \times n$ Latin Square (which is adapted from the Sudoku encoding by removing the square constraints), we vary the parameter n and measure the learning time on the random examples (in equal quantities). In Figure 1b, we again observe a linear behavior. The slope depends on the size of an example, which seems to be logical because the number and the size of examples directly affects the grounding size. Interestingly, the quadratic growth of an example size shows only moderate effect on the linear slope of the behavior.
To address $\mathbf{Q}_3$, we split the calls to Gringo and Clasp and we have measured the time it takes for each of the systems separately. As Figure 2b indicates the runtime for both systems behaves similarly, while staying approximately linear with the size of the input. This might be cause, for example, the internal procedures such as building the datastructures to ground and solve the problem, the solver needs to analyze and index all the examples it is given (each Sudoku example is 81 atoms, therefore 300 examples is around 25k atoms); more clearly it indicates that there is no significant jump or blow-up in the grouding/solving time with respect to the growth of the input size.
To address $\mathbf{Q}_4$, we took multiple publicaly available models of the N-queens problem and measure its runtine on the randon examples. While models show certain diversity in the runtime, they seem to follow pattern and do not demonstrate any significant deviations, e.g., of an order of magnitude.

3 Grounding estimate

Predicate reification grounding We consider the general case of sketching a custom predicate $?p/n$ here:

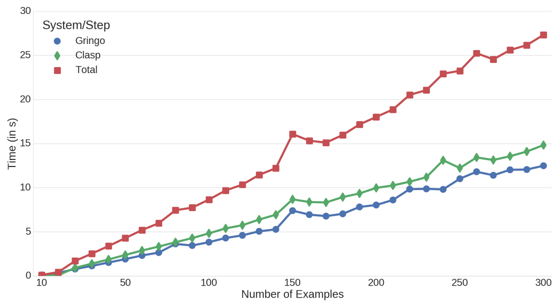
$$?p/n \mapsto p_1/n, \ldots, p_k/n \tag{1}$$

The ASP code generated for it would look as follows:

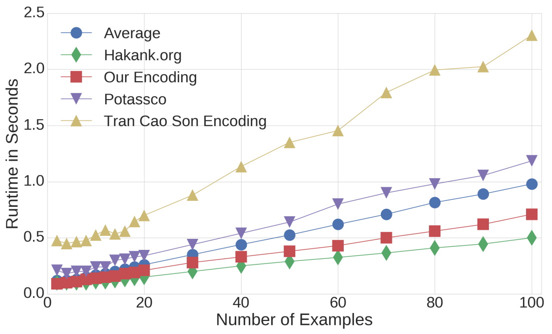
```
p(1, X1, ..., Xk) :- p1(X1, ..., Xk).
                ...
p(k, X1, ..., Xk) :- pk(X1, ..., Xk).
```

Then for `p/n+1` (it has an extra index variable) a grounder, like Gringo, would generate the grounding of the size:

$$|grounding(p)| = \sum_{i=1}^{k} |grounding(p_i)|$$



(a) $\mathbf{Q}_3$: investigating the runtime distribution on Sudoku between Gringo (grounding) and Clasp (solving) separately



(b) $\mathbf{Q}_4$: investigating the effect of encoding variations on runtime; N-queens

Figure 2: Investigating $\mathbf{Q}_3$ and $\mathbf{Q}_4$: separate grounding/solving measurments (left) and effect of the encoding variations on runtime (right; N-queens)

where $|grounding(\dots)|$ is the size of grounding of the given program.

Example extension grounding If a predicate p_i depends on the example predicates, that its grounding $grounding(p_i)$ also depend on the examples' grounding, namely, the predicate p_i gets extended as

```
p_i(e_index, c_1,..., c_k)
```

where `e_index` is the constant representing the example index and `c_j` are constants specified in (or inferred from) the examples, i.e.,

$$|grounding(p_i)| = \sum_{e_j \in E^+ \cup E^-} |grounding(e_j)|$$

This allow us to estimate how the grounding of a particular predicate changes in the sketched setting. Furthermore, on the rule level we introduce additional atoms, let us indicate much of the grounding overhead it adds on top of grounding without sketching.

Additional atoms in the rules Our estimate on the growth of atoms due to example dependency already gives a correction on the example atom introduced into the rules, since example variables have always the same name, so they are always bound together (and the grounders like Gringo take advantage of that). As well adding decision atoms does not increase the grounding, simply because they are always bound to the sketched predicates, i.e., they have the same name as the index variable of p.

Decision choice rules Adding additional rules of course adds to the grounding of the program. If we look into Eq. 1, the corresponding decision choice rule would look like:

```
1 { decision_p(X) : choice_p(X) } 1.
```

which is to the best of our knowledge, results, in the grounding linear in the size of `choice_p(X)`, i.e., it is linearly proportional to the k, the number of sketched choices of $?p$. Since, the decision rules are independent of each other, their grounding sums up.

Rule split grounding Since each the split operation introduces negative and positive version of a rule r, then the overall grounding overhead is $2 \times grounding(r)$, since the example index variables in the atoms *negative* and *positive* is already bound and accounted for in the example step.

Grounding of special cases For the sketched inequalities, arithmetic and negation the grounding estimate is similar, since their domains must be explicitly specified in the sketch.

Table 2: Key feature comparison between SkASP and ILASP

Feature	SkASP	ILASP
Problem type	Enumeration	Optimization
Input	Sketch, examples, preferences	Modes, examples
Intermediate	ASP code (+post-processing)	Sets of positive & violating solutions
Output	A set of preferred solutions	A minimal in size program
Background knolwedge	Yes	Yes
Single shot	Yes	No
Local uncerainty	Yes	No
Supports choices	No	Yes
Meta-ASP rewriting	Yes	Yes

4 Feature comparison with ILASP

We indicate in Table 2 the key feature comparison of SkASP with the inductive ASP learning system ILASP.

5 Preferences handling

The preferences are indicated in the sketch with a keyword `[PREFERENCES]` and define a partial order on the solutions.
For example, if $?p$ can be mapped to $p_1, \dots, p_k$ with preferences $w_1^p, \dots, w_k^p$ (where each w_i^p is an integer) and q correspondinly to $q_1, \dots, q_h$ with $w_1^q, \dots w_h^q$, then if a substitution $p \mapsto p_i, q \mapsto q_j$ is a solution, it has a corresponding preference tuple (w_i^p, w_j^q), then this solution is *preferred* if there is no other solution substitution $p \mapsto p_f, q \mapsto q_g$ with the preference tuple (w_f^p, w_g^q) such that $w_f^p > w_i^p \wedge w_g^q \geq w_j^q$ or $w_f^p \geq w_i^p \wedge w_g^q > w_j^q$, i.e., if the preference tuple (w_i^p, w_j^q) of the solution is Pareto-optimal.
This Pareto-optimality introduces a partial order on the set of preferred solutions.
While the number of potential Answer Sets is generally exponential for a sketched ASP, the number of programs actually satisfying the examples is typically rather small (10000-20000). If that is not the case, then the problem is usually underconstrained and it needs more examples, no user would be able to go over a million of proposed programs. Consequently, we currently implement the preference handling as a post-processing mechanism, which also allows to apply our method to Constraint Programming and Constraint Logic Programming.

Implementation details An overview of the implementation to post-process the solutions is given in Algorithm 1. Simply speaking, it iterates over the pairs of solutions and checks pairwise dominance, to optimize the iteration, we only consider each pair once and maintain a set of dominated solutions that can be skipped and do not need to be checked, here we use the transitive property of the Pareto dominance, if θ

dominates θ', then θ also dominates θ'', provided that θ' dominates θ''.

```
Data: A set of solutions $O$, preferences $f$
Result: A set of preferred solutions $S$
$dominated \leftarrow \emptyset$
for $o \in O$ do
    if $o \in dominated$ then
        continue
    end
    for $o' \in O$ do
        if $index(o) > index(o')$ or $o \in dominated$ then
            continue
        end
        $update\_dominance(o,o',f,dominated)$
    end
end
$S \leftarrow O \setminus dominated$ return $S$
```

Algorithm 1: Preference handling: post-processing implementation overview

6 Application to Query-by-Example

SQL is a powerful formalism for querying relational databases. However, writing and debugging SQL queries is far from trivial. Existing debugging tools are designed for advanced users and while they guide in locating the bugs, they provide little help in correcting them. Contrary to this approach, query-by-example lets a user specify examples and generates a query that satisfies them. What is missing is the ability to combine these two approaches, that is to partially specify a query and leave some others parts open and to be completed using some examples of the desired outputs of the query.

Consider a standard database setting with an employee database, where a user needs to select all highly-paid agents who are not involved in sales. The user knows the general structure of the query, i.e. a join between two tables with a filtering condition on corresponding columns, but she is unsure of the exact values or combination of values of the columns. She thus leaves all the comparisons with constants open. Furthermore, she knows that "Peter" is in the answer while "Mary" is not. The beauty of this is in the combination of the user's knowledge of the query with a straightforward way to specify uncertainty in the query.

SQL representation

```
SELECT Fname, Lname
FROM Emp as e, Dept as d
WHERE e.d_id = d.id
AND e.pos         = ?c1
AND e.salary      > ?c2
AND d.dept_funct != ?c3
```

Datalog representation

```
q(Fname,Lname) :-
   emp(Id,Did,Fname,Lname,Pos,Salary),
   dept(Did,Dname,Function,Manager),
   Pos       = ?c1,
   Salary    > ?c2,
   Function != ?c3.
```

Solving Query Sketching Conceptually the algorithm to solve Query Sketching can be described as *Query Optimization + Search.*

The key motivation for query optimization is that modern constraint solvers cannot handle smoothly millions or even hundreds of thousands of records, but they are particularly good when the problem has a number of constraints that propagate well. It is thus reasonable to reduce the initial search space by using the problem structure. We propose a number of query optimization techniques that aim at supporting the search engine (ASP) in finding the solutions and avoiding doing unnecessary grounding.

The search part is to find a substitution θ for the sketched variables such that when applied to the query, all positive and none of the negative examples are satisfied. To do so, we need to search in the space of sketched constants. This part can be naturally implemented using ASP sketching techniques.

This application of SkASP in the database context can be seen as an intelligence assistant in writing queries and due to wide spread of SQL databases can find its users among people learning or debugging SQL queries.

7 Unstratified Case

According to the definition of sketching, the sketched program must have no answer set for any of the negative examles.
Then, if we take an ASP program encoding graph coloring (choice rules can be encoded using unstratified negation) without any sketched variables and give as the negative example the correct coloring, i.e., r-b, r-g, b-g pairs, the sketched problem is satisfied by an empty substitution iff there is no solution satisfying to the coloring schema, i.e., it is a solution to a CoNP-complete problem. This indicates that there is a complexity jump in the unstratified setting.
We are grateful to _____ for indicating the complexity jump in the unstratified case.

8 Input, Intermediate-Step, Output Examples

Input Sketch We illustrate the input-intermediate-output steps using the problem of subgraph isomorphism.
We are given a sketch of the constraints of the problem, together with auxiliary predicates and three example, followed by the domains of sketched variables, a set of facts and the default preferences (equal and not equal are preferred over all inequalities).

```
[SKETCH]
:- map(X,N) & map(Y,M) & X ?= Y & N ?= M.
:- map(X,N) & map(Y,M) & ?p1(X,Y) & ?not ?p2(N,M).

%auxilary predicates
edge1(Y,X) :- edge1(X,Y).
edge2(Y,X) :- edge2(X,Y).

[EXAMPLES]
positive: map(d,1). map(b,3). map(a,5). map(c,4). map(e,6).
negative: map(d,3). map(b,1). map(a,5). map(c,4). map(e,6).
negative: map(d,1). map(b,1). map(a,1). map(c,1). map(e,1).

[SKETCHEDVAR]
?p1/2 : edge1, edge2
?p2/2 : edge1, edge2

[FACTS]
edge1(a,b). edge1(a,e). edge1(b,c). edge1(b,d). edge1(d,c). edge1(d,d).
edge2(1,1). edge2(1,2). edge2(1,3). edge2(1,4). edge2(2,4). edge2(3,4).
edge2(3,5). edge2(5,6).

[DOMAIN]
?= : 1,2,3,4,5,6,a,b,c,d,e
not : 1,2,3,4,5,6,a,b,c,d,e

[PREFERENCES]
?= : = -> max, != -> max.
```

Generated ASP This ASP code is generated internally and not meant to be modified by a human.

```
%%%%% ORIGINAL SKETCH RULES %%%%%%
%       :- map(X,N),map(Y,M),X ?= Y,N ?= M.
%       :- map(X,N),map(Y,M),sketched_p1(X,Y),sketched_p2(N,M).
%%%%% DOMAIN AND GENERATORS %%%%%%
sketched_clauses(1..2).equalities(eq). equalities(eq).  equalities(leq).
equalities(geq). equalities(le). equalities(ge). equalities(ne). equalities(unbound).
sketched_p1_choice(c_edge1).sketched_p1_choice(c_edge2).
sketched_p2_choice(c_edge1).sketched_p2_choice(c_edge2).

1 { decision_eq_1(X) :  equalities(X) } 1.
1 { decision_eq_2(X) :  equalities(X) } 1.
1 { decision_sketched_p1(X) :  sketched_p1_choice(X) } 1.
1 { decision_sketched_p2(X) :  sketched_p2_choice(X) } 1.

%%%%%% KEEP DOMAIN INFORMATION ABOUT DECISION VARIABLES %%%%%%
domain_not(X) :- sketched_p1_choice(X).
domain_not(X) :- sketched_p2_choice(X).

%%%%% NON-EXAMPLES FACTS %%%%%%
edge1(a,b). edge1(a,e). edge1(b,c). edge1(b,d).
edge1(d,c). edge1(d,d). edge2(1,1). edge2(1,2).
edge2(1,3). edge2(1,4). edge2(2,4). edge2(3,4).
edge2(3,5). edge2(5,6).
%%%%%%  MATERIALIZED EQUALITY %%%%%%
          eq(eq , X, Y) :- X == Y, domain0(X), domain0(Y).
          eq(leq, X, Y) :- X <= Y, domain0(X), domain0(Y).
          eq(geq, X, Y) :- X >= Y, domain0(X), domain0(Y).
          eq(le , X, Y) :- X <  Y, domain0(X), domain0(Y).
          eq(ge , X, Y) :- X >  Y, domain0(X), domain0(Y).
          eq(ne , X, Y) :- X != Y, domain0(X), domain0(Y).
          eq(unbound, X, Y) :- domain0(X), domain0(Y).

%%%%%% SET DOMAIN VALUES %%%%%%
domain0(1). domain0(2). domain0(3). domain0(4). domain0(5). domain0(6).
domain0(a). domain0(b). domain0(c). domain0(d). domain0(e).

examples(E) :- negative(E).
examples(E) :- positive(E).
domain_not_examples(E) :- examples(E).
sketched_p1(c_edge1,X0,X1) :- edge1(X0,X1).
sketched_p1(c_edge2,X0,X1) :- edge2(X0,X1).
sketched_p2(c_edge1,X0,X1) :- edge1(X0,X1).
sketched_p2(c_edge2,X0,X1) :- edge2(X0,X1).

%%%%% NON-SKETCHED INFERENCE RULES %%%%%%
edge1(Y,X) :- edge1(X,Y),examples(Eaux).
edge2(Y,X) :- edge2(X,Y),examples(Eaux).
%%%%% FAILING RULES %%%%%%
fail :- not neg_sat(E), negative(E).
:- fail.
%%%%% SKETCHED INFERENCE RULES %%%%%%
%%%%% POSITIVE SKETCHED RULES %%%%%%
fail :- map(Eaux,X,N),map(Eaux,Y,M),eq(Q_eq_1,X,Y),eq(Q_eq_2,N,M),
                      decision_eq_1(Q_eq_1),decision_eq_2(Q_eq_2),positive(Eaux).
fail :- map(Eaux,X,N),map(Eaux,Y,M),sketched_p1(Q_sketched_p1,X,Y),
        sketched_not_1(Not_D1,Q_sketched_p2,N,M),decision_sketched_p1(Q_sketched_p1),
        decision_sketched_p2(Q_sketched_p2), positive(Eaux),decision_not_1(Not_D1).
%%%%% NEGATIVE SKETCHED RULES %%%%%%
neg_sat(Eaux) :- map(Eaux,X,N),map(Eaux,Y,M),eq(Q_eq_1,X,Y),eq(Q_eq_2,N,M),
                 decision_eq_1(Q_eq_1),decision_eq_2(Q_eq_2),negative(Eaux).
neg_sat(Eaux) :- map(Eaux,X,N),map(Eaux,Y,M),sketched_p1(Q_sketched_p1,X,Y),
                 sketched_not_1(Not_D1,Q_sketched_p2,N,M),decision_sketched_p1(Q_sketched_p1),
                 decision_sketched_p2(Q_sketched_p2),negative(Eaux),decision_not_1(Not_D1).
%%%%% DECISION ON THE NEGATION %%%%%%
1 { decision_not_1(positive); decision_not_1(negative) } 1.
%%%%% REIFICATION OF THE NEGATION %%%%%%
sketched_not_1(positive,Q_sketched_p2,N,M) :- sketched_p2(Q_sketched_p2,N,M).
sketched_not_1(negative,Q_sketched_p2,N,M) :- not sketched_p2(Q_sketched_p2,N,M),domain_not(N),
                              domain_not(M),domain_not(Q_sketched_p2),domain_not_examples(Eaux).
%%%%% DOMAIN OF THE NEGATION %%%%%%
domain_not_example(Eaux) :- examples(Eaux).
domain_not(1). domain_not(2). domain_not(3). domain_not(4). domain_not(5).
domain_not(6). domain_not(a). domain_not(b). domain_not(c). domain_not(d). domain_not(e).
%%%%% EXAMPLES %%%%%%
positive(0). map(0,d,1). map(0,b,3). map(0,a,5). map(0,c,4). map(0,e,6).
negative(1). map(1,d,3). map(1,b,1). map(1,a,5). map(1,c,4). map(1,e,6).
negative(2). map(2,d,1). map(2,b,1). map(2,a,1). map(2,c,1). map(2,e,1).
#show neg_sat/1.
#show decision_eq_1/1.
#show decision_eq_2/1.
#show decision_sketched_p1/1.
#show decision_sketched_p2/1.
#show decision_not_1/1.
```

Output Constraints That is an example of the output, it first prints the decisions made and then the generated programs, in this particular example there is only one solution and one program is generated.

```
Decision prints
before dominance models: 3
solution # 0
decision_eq_1 is ['ne']
decision_eq_2 is ['eq']
decision_not_1 is ['negative']
decision_sketched_p1 is ['c_edge1']
decision_sketched_p2 is ['c_edge2']
after dominance models: 1

Generated Programs:
Solution 1
          :- map(X,N), map(Y,M), X != Y, N = M.
          :- map(X,N), map(Y,M), edge1(X,Y), not edge2(N,M).
edge1(Y,X) :- edge1(X,Y).
edge2(Y,X) :- edge2(X,Y).
```