Automatic generation of English-language steps in puzzle solving

Hemantha Ponnuru, Raphael Finkel, Victor Marek, and Mirosław Truszczyński
Computer Science Department
University of Kentucky
Lexington, KY 40506
{hkponn2,raphael,marek,mirek}@cs.uky.edu

Abstract

This paper shows how to generate an English step-by-step explanation that describes how an automated reasoning system solves a complex constraint program. We study programs that solve tabular-constraint problems encoded in Constraint Lingo and then translated into propositional logic with cardinality constraints. We instrument the logic solver so that it generates log files. We inspect those logs and express reasoning steps in English by using grammatical information placed in the Constraint Lingo program.

Keywords: logic puzzles, logic programming, constraint satisfaction, tabular constraint satisfaction

1. Introduction

In declarative programming, programmers build theories to encode problems and then apply automated reasoning techniques to generate solutions. The question we study here is generating explanations of the steps leading to these solutions and presenting them in human-readable format. We demonstrate our approach in the setting of logic-puzzle solving, but it applies equally well to other applications.

We start with an example [1]:

> Five senior citizens, Alicia, Bert, Cecil, Dexter, and Edith, became famous for some activity. Their last names (in no particular order) were Foster, Garland, Hollis, Izenberg, and Johnston. They were (in no particular order) 101, 103, 106, 108, and 111 years old. The activities were going on a safari, making a sports video, throwing the first pitch at a baseball game, participating in a game show, and skydiving. Match the first and last names to the age and activity.
>
> 1. Bert, Cecil, and Dexter were male; Alicia and Edith were female.
> 2. Izenberg did not go on a safari.
> 3. Dexter was younger than the one who threw the first pitch but older than Mr. Foster.
> 4. Alicia, who was not Hollis, was 111 years old.
> 5. Dexter, who is neither Garland nor Hollis, participated in a game show.
> 6. Izenberg, who was not 101 years old, participated in skydiving.
> 7. Dexter was younger than Cecil but older than Izenberg.

To solve a logic puzzle, we (1) encode it in the high-level language Constraint Lingo [3, 4], (2) mechanically translate this representation to a theory in propositional logic with cardinality constraints [1], and (3) compute models of that theory by means of a search procedure that relies of unit propagation and branching. Models of the theory correspond to solutions of the puzzle.

Our computation shows that this logic puzzle has exactly one solution:

first	activity	age	gender	last
alicia	ashow	111	female	garland
bert	skydiving	101	male	foster
cecil	pitch	108	male	hollis
dexter	safari	106	male	johnston
edith	video	103	female	izenberg

The trace of the search procedure provides a formal proof that answers are indeed models and that no other models exist. Our goal is to develop techniques to collect

This work was partially supported by NSF grants 0097278 and 0325063.

[1] Based on the puzzle "100 and then some," Dell Logic Puzzles June 1999, page 21; used by permission.

the trace and transform it into human-readable format. To this end, we (1) expand Constraint Lingo so the programmer can embed linguistic information concerning the puzzle, (2) modify translators from Constraint Lingo to propositional logic to retain necessary information, (3) modify the solver to log all propagation and backtracking steps, (4) develop tools to generate English text from the Constraint Lingo program and the log.

These steps lead to an explanation of the reasoning that leads the solver to the solution. An excerpt of that explanation is shown here. We indent a step deeper than its predecessor to indicate a new line of reasoning. We indent a line more shallowly than its predecessor to indicate a conclusion based on all the preceding lines at the previous indentation level.

```
      (18) Clue 3 tells us Dexter isn't
      101 years old.
         We know (step 16) that Senior
         Izenberg isn't 101 years old
      (19) Clue 7 tells us Dexter isn't
      103 years old.
   (20) Clue 7 tells us Cecil isn't 106
   years old.
      We know (step 4) that Alicia is 111
      years old.
   (21) By elimination, Cecil isn't 111
   years old.
(22) By elimination, Cecil is 108 years
old.
```

2. Constraint Lingo

We translate puzzles into our Constraint Lingo language for tabular-constraint problems [3, 4]. This language lets us capture the constraints without settling on any particular logic formalism or any coding strategy within that formalism.

We encode our puzzle by the following Constraint Lingo code.

```
CLASS first: alicia bert cecil dexter
    edith
CLASS last: foster garland hollis izenberg
    johnston
CLASS age: 101 103 106 108 111
CLASS activity: safari video pitch ashow
    skydiving
PARTITION gender: male female

AGREE male: bert cecil dexter # clue 1
AGREE female: alicia edith # clue 1

CONFLICT safari izenberg # clue 2

BEFORE age: dexter pitch # clue 3
BEFORE age: foster dexter # clue 3
AGREE male: foster # clue 3

CONFLICT alicia hollis # clue 4
REQUIRED alicia 111 # clue 4
AGREE male: foster # clue 4

CONFLICT dexter garland # clue 5
CONFLICT dexter hollis # clue 5
REQUIRED garland ashow # clue 5

CONFLICT 101 izenberg # clue 6
REQUIRED 101 skydiving # clue 6

BEFORE age: dexter cecil # clue 7
BEFORE age: izenberg dexter # clue 7
```

This example shows the most important constructs of Constraint Lingo. **Columns** are defined by `CLASS` or `PARTITION`; the former implies a used-once constraint, but the latter does not. **Row references** are members of a `CLASS`, such as `dexter`; they refer to the one row that contains that member. **Constraints** involve rows. `REQUIRED` constrains the rows specified by the given row references to be the same row. `CONFLICT` constrains the rows specified by the given row references to be distinct rows. `AGREE` constrains the specified rows to have the given `PARTITION` member in the appropriate column. (These three constructs allow more than two row references.) `BEFORE` constrains the values of a numeric `CLASS` (here, `age`) to have a given order in two specified rows.

Constraint Lingo includes other constructs not needed for this example. **Columns** can also be defined by `MAP`, which indicates a relation among rows, which can optionally be constrained to be onto, nonreflexive, symmetric, or asymmetric. For instance, if every senior had a friend (another senior), we would introduce a `MAP` column called `friend` and constrain it to be nonreflexive. Columns may also be defined by `POWERCLASS`, which constrains its entries to be subsets (not just singletons) of its members, with cardinality optionally restricted both above and below. A numeric column may be underspecified, which means it has more members than there are rows; only some of those members are used in any solution. A numeric column may also be marked `circular`, which affects ordering constraints. **Row references** can also be introduced variables, which are constrained to refer to exactly one row. Constraint Lingo

also allows set-theoretic combinations of row references. **Cell references** refer to a particular column of a specified row, such as `first(skydiving)`. **Constraints** include `MATCH`, which equates two sets of rows, `SAME` and `DIFFER`, which constrain two rows to agree or disagree on the value of a `PARTITION` column, and `IMPLIES`, which constrains the specified rows to include a specified `POWERCLASS` member in its column. `ORDER` constrains the values in a numeric column to satisfy an additive or multiplicative constraint. `DIFFER` constrains two `PARTITION` columns so that any two rows show a different combination of members in those columns. `USED` bounds above and below the number of times a `PARTITION` element is used in its column. `DIFFER` constrains two `MAP` columns to differ everywhere. `AGGREGATE` constrains either the sum or the product of the elements in an underspecified numeric `CLASS` column. `SET` constrains the values of a cell (given by a cell reference) or numerically relates the values to two cells.

3. Translation into aspps

We have built translators from Constraint Lingo into several logic formalisms. The results presented in this paper are based on translation into propositional logic with cardinality constraints, for which we use the *aspps* solver [1][2]. At the heart of our translation are **cross-column predicates** for every pair of columns, such as `first_age(·,·)`. The translation includes rules that constrain `CLASS` columns to use each appropriate member only once. Each Constraint Lingo constraint translates into one or more additional rules. For example, our translation for `CONFLICT safari izenberg` has one rule. Its empty head means "must not occur".

```
activity_last(safari,izenberg) -> .
```

As a second example, `BEFORE age: dexter cecil` translates into two rules:

```
age_first(Age, dexter),
    age_first(Age1, cecil),
    Age >= Age1 -> .
age_first(111, dexter) -> .
```

The solver uses unit propagation and backtrack to determine which predicates are true (such as `first_age(dexter,106)`) and which are false (such as `first_age(dexter,108)`). In general, we could apply our reasoning explainer to any *aspps* program. The solution is the set of true predicates.

[2]The *aspps* solver has two phases. The first converts rules into ground instances and applies unit propagation to discover direct consequences. The second hypothesizes truth values and backtracks to find consistent models. We treat these two phases together for this discussion.

4. Recovering the reasons

We analyze the log generated by the solver to reconstruct its reasoning. Unit propagation derives true predicates in three ways. (1) The predicate is a fact. The reason is then just the fact. (2) The predicate is the conclusion (head) of a rule whose body predicates are all true. The reason is this rule along with the set of reasons for each of those body predicates. (3) The rule is a cardinality constraint with at least n solutions, and all but n of the predicates are false. The reason is this rule along with the set of reasons that those other predicates are false.

Unit propagation derives false predicates in three ways. (1) The rule has an empty head (it is a failure rule), and all but one predicate in the body has been shown to be true. The remaining body predicate must be false; the reason is this rule along with the set of reasons that the other predicates are true. (2) The rule has a head that is a predicate already shown to be false, and all but one predicate in the body has been shown to be true. The remaining body predicate must be false; the reason is this rule along with the set of reasons that the other body predicates are true and the reason that the head predicate is false. (3) The rule is a cardinality constraint with at most n solutions, and n predicates have already been shown to be true. The remaining predicates are false; the reason is this rule along with the reasons for each of those n true predicates.

Backtracking involves guessing the truth value of a predicate. Failure returns to the previous guess and derives the opposite of that guess.

5. Translating the reasons into English

Explanations need to say that a cross-column predicate is true or false. We state the truth of `age_first(111, dexter)` as "Dexter was 111 years old", and the falsity of `activity_age(skydiving,101)` as "the senior who was 101 years old was not recognized for skydiving."

In order to do so, we need to know the relationships among the various columns. In our puzzle, the following hierarchy applies:

`first/last (age activity gender).`

That is, a person is identified by a first or last name and has properties of age, activity, and gender. The program-

mer must indicate the hierarchy as part of the Constraint Lingo program.

We then translate `age_first(111,dexter)` by saying that `dexter` has property `111`, not the reverse, because `first` is above `age` in the hierarchy. For `activity_age(skydiving, 101)`, both `activity` and `age` are subordinate to `first`, so neither is central. We introduce a reference to the person who has both these properties, "the senior who."

Two levels of hierarchy are usually enough. However, the following is a three-level hierarchy.

```
town (founded state motel
    (street john alice))
```

This example comes from a puzzle where motels are visited by John and Alice in some order; each motel is in a town founded in some year in some state and on some street. We express `founded_state(1888,tennesee)` by referring to the town that connects them: "the town that was founded in 1888 is in Tennessee." We express `john_state(7,ohio)` by referring to the town and the motel: "the town in Ohio has the motel that was visited 7th by John."

Sometimes a class is implied by the puzzle but is not part of its description. For instance, the "Field Trips" puzzle[3] involves children with teachers who hold sessions on particular days at particular sites, leading to declarations like these:

```
CLASS child: chester jeanette marvin
CLASS day: monday tuesday wednesday
CLASS teacher: brandon dempsey lewis
CLASS site: cloisters guggenheim liberty
```

We let the programmer introduce a virtual class into the hierarchy as a placeholder for a class that could have been part of the puzzle but isn't:

```
child (teacher session (day site)).
```

The virtual class `session` lets us generate statements like "Jeanette attended the session on Wednesday" and "the child taught by Brandon attended the session at the Guggenheim Museum."

In addition, the programmer specifies four grammatical aspects of each class.

Subject: how to express an element (like `dexter`) as the subject of a sentence:

```
first:  %M
last:  Senior %M
```

`%M` is a format string meaning the member value (such as `dexter`) with the first letter capitalized (`Dexter`). The format for `last` adds an honorific, turning the member value `johnston` into `Senior Johnston`.

Predicate: how to express an element (like `age`) as the predicate of a sentence:

```
last:  Senior %M
age:  %m years old
activity:  %m
gender:  %m
```

Verb: the positive and negative verbs to govern an element in the predicate:

```
last:  was, wasn't
age:  was, wasn't
activity:  was recognized for,
   was not recognized for
gender:  is, isn't
```

Relative: how to introduce a reference to a class that connects two other classes:

```
first:  the senior who
last:  the senior who
```

We let the programmer describe proper English expansions for one-word shorthands:

```
ashow: participating in a game show
pitch: throwing the first pitch at a
    baseball game
video: making a sports video
safari: going on a safari
```

We omit here the precise format by which the programmer specifies the hierarchy of classes, the four grammatical aspects of each class, and one-word shorthands.

6. Presenting the reasons

We don't present all the steps that the automatic reasoning system takes; instead, we show only positive conclusions. However, if a positive conclusion requires as a reason a negative conclusion, then we show that, too. Our log lets us build a DAG whose nodes represent conclusions and whose edges represent reasons. As we build that graph, we output an English explanation for each positive node. This explanation may depend on antecedents, positive or negative. If an antecedent has been explained in an earlier step, we just say "We know (step n) that . . .". We recursively explain other antecedents.

Often a conclusion is based directly on a clue, in which case we cite that clue. Other conclusions are based on cardinality constraints, in which case we say "By elimination, . . .". Still other conclusions are are applications of transitivity: If Foster is recognized for skydiving, and

[3] Dell Logic Puzzles, June 1999, page 15

the 101-year-old was recognized for skydiving, then Foster was 101 years old. In such cases we say "To be consistent, . . .".

When the solver introduces a guess, we say "Assuming . . .". A guess may lead to failure, which happens when the solver derives both a fact and its negation. We then say "Step n and step m contradict each other, so . . .". Several guesses may be listed in one assumption; We only list those guesses G that are actually used in explaining a fact F, even though the solver might have made additional assumptions at the point it got to this step. Later, if the solver is using a set of guesses $G' \supseteq G$ and it again proves F, we need not repeat the explanation, even though the *aspps* solver sometimes does repeat its reasoning or even discover an independent proof of F. Instead, we just refer to the previous step.

7. Discussion

To our knowledge, this work is the first success at generating natural-language reasoning steps by instrumenting a general-purpose solver, although the concept of explaining reasons has a history reaching back to the Mycin project [7].

General-purpose solvers don't follow the same path that a human would. The *aspps* solver does not remember the facts it has derived, leading it to often rederive the same fact. Subsequent derivations can be longer or shorter than the first one. We display the first derivation and refer back to it if the same fact is needed later. Our explanations would be shorter if we searched the log for the most concise derivation for each fact.

Furthermore, the length of explanations indicates that *aspps* does not find the simplest reasoning path. For instance, the explanation for the 100 puzzle requires about 150 steps. The published explanation is significantly shorter, partly because it uses shorthands that expand to several steps.

We see several benefits of our approach. First, it shows how to explain, in nontechnical language, solving a logic puzzle.

Second, our results can be generalized to other logic programs. We only require that the solver log its steps and that the logic program include comments referring to clues and showing grammatical relationships. We can explain how to solve any logic program whose solution is based on a combination of unit propagation and backtracking. We can generate the explanation automatically (with some grammatical assistance from the programmer), and we can do so in the presence of cardinality constraints.

Third, our work produces convincing arguments supporting the output of the solver. The explanations are "locally checkable", in that an interested reader can follow any set of steps, even though few readers would want to scrutinize the entire explanation.

Fourth, our approach leads to non-textual representations of the reasoning steps. Our software can present the steps in a graphical format, which is often more intuitive than text. The argument takes the form of a tree. Each node on the path from the root to the leftmost leaf represents a guess that turns out to be true. Each guess has a sibling that represents its negation. These siblings always lead to contradictions. We merge subtrees that prove the same results in a similar context (that is, under a superset of the same assumptions). Generating these trees in a readable and compact form is challenging.

Fifth, our work with explaining the reasoning in solving puzzles has led us to better *aspps* code for Constraint Lingo programs. There are occasions when *aspps* takes many steps where a human would take only one. The reason for this discrepancy is that our translation from Constraint Lingo is often too "spare", containing only enough rules to force a unique, correct answer. Additional rules, although not necessary for correctness, allow for more direct solution. For example, Clue 7 tells us that Dexter was younger than Cecil. We can generate an additional rule saying that if Dexter was 106, then Cecil must be 108 or 111. When we add such additional rules, we find that *aspps* is able to derive many more facts by unit propagation, leaving fewer cases where it needs to guess. So inspecting the proofs has led to better translation of Constraint Lingo.

Sixth, explanations of models can be quite important. Although we have focused on the explaining solutions to logic puzzles, the problem of finding explanations is not limited to recreational mathematics. We illustrate this point with two examples from different application domains.

Consider the domain of combinatorial optimization. As we have pointed out [4], Constraint Lingo lets us represent complex combinatorial problems such as graph coloring, vertex cover, and Hamiltonian cycles. Let's focus on the Hamiltonian-cycle problem. As the solver searches for a solution, it performs several types of reasoning: (1) guessing that an edge belongs to the putative cycle, (2) checking that each vertex contributes to two edges in the cycle, (3) deriving that a given edge must belong to the cycle because of cardinality constraints, requirements, (4) testing that a cycle has been constructed,

and (5) testing that all vertices are in the cycle. Our techniques can produce an English explanation that describes the guesses and the graph properties that lead us to the solution. If the graph represents a communication network or some other real-life entity, we may gain valuable information about the topology of that network.

Another domain is bounded model checking. We may want to represent a nondeterministic finite automaton as an instance and describe the trajectories of that machine as solutions to a Constraint Lingo program. It is possible to describe bounded-length trajectories of a nondeterministic automaton in a logic formalism so that trajectories correspond to solutions [5]. If the solver returns a path leading to an undesirable state, we may want to find reasons that lead to such a solution. The reasons might be particular choices, or they may be consequences of constraints that describe the machine. The reasons can help the designer of the machine debug the physical realization of the automaton.

8. Software

The suite of programs for generating, translating, solving, and explaining Constraint Lingo programs is freely available at

`ftp://ftp.cs.uky.edu/cs/software/cl.tar.gz`

This suite contains over 170 sample Constraint Lingo programs encoding a wide range of logic puzzles. Targeted solvers include not only *aspps* but also *smodels* [6], *dlv* [2], and ECL$^{\text{i}}$PS$^{\text{e}}$ [8].

9. References

[1] D. East and M. Truszczyński. *aspps* — an implementation of answer-set programming with propositional schemata. In *Proceedings of Logic Programming and Nonmonotonic Reasoning Conference, LPNMR 2001*, volume 2173, pages 402–405. Lecture Notes in Artificial Intelligence, Springer Verlag, 2001.

[2] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A KR system `dlv`: Progress report, comparisons and benchmarks. In *Proceeding of the Sixth International Conference on Knowledge Representation and Reasoning (KR '98)*, pages 406–417. Morgan Kaufmann, 1998.

[3] R. Finkel, V. Marek, and M. Truszczyński. Tabular constraint-satisfaction problems and answer-set programming. *AAAI-2001 Spring Symposium Series, Workshop on Answer Set Programming*, 2001.

[4] R. Finkel, V. Marek, and M. Truszczyński. Constraint lingo: Towards high-level constraint programming. *Software Practice and Experience*, to appear, 2004.

[5] K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, Vienna, Austria, September 17-19, 2001.

[6] I. Niemelä and P. Simons. Extending the smodels system with cardinality and weight constraints. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer Academic Publishers, 2000.

[7] Edward Shortliffe. *MYCIN: Computer-Based Medical Consultations*. American Elsevier, 1976.

[8] M. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A platform for constraint logic programming, 1997. `http://www.icparc.ic.ac.uk/eclipse/reports/eclipse.ps.gz`.