Rule-based solution to Crypto-Arithmetic Problems

Lorenzo Ciampiconi
A0179901X
National University of Singapore,
Exchange Students from
Politecnico di Milano, Italy

Abstract

The science of solving cryptarithms is called crypto-arithmetic. A cryptarithms is a matematical puzzle represented by addition of literals to be solved substituting to the literals a unique digit in the way to obtain a valid arithmetic addition.
A rule-based system can encode and generate solution for this kind of problem with the definition of facts and rules to be fired, this work can help to understand the relation between human thinking and expert system and how is possible to design systems that mimic the human behavior while approaching this problem.

Constraints and Results

Definitions

- A is the set of variables given in input
- $\vec{op}_i \mid \forall i \in [1,2]$ is a operator of the addition and op_{ij} is the j-sim variable of $\vec{op}_i$
- $\vec{res}$ is the result of the addition and res_{ij} is the j-sim variable of $\vec{res}$
- $\forall x \forall i (\in [1,2] \in \vec{op}_i) \Rightarrow x \in A$
- $\forall x \in \vec{res} \Rightarrow x \in A$
- $\vec{\sigma_i}$ is a solution, σ_{ij} is an assignment to $x \in A$
- $\forall i |\vec{\sigma_i}| = |A|$ (a solution is defined when it contains an assignment for all the variables)

Constraints

The solution proposed has the following constraints for the input and the solution given:

- $|A| \leq 10$
- $|op_1| = |op_2|$
- $|res| = |op_1| \vee |res| = |op_1| + 1$
- $\forall i \; \forall j, k \in [1, |\vec{\sigma_i}|] \; j \neq k \Leftrightarrow \sigma_{ij} \neq \sigma_{ik}$
- $\forall i \; \forall j \in [1, |\vec{\sigma_i}|]$ if σ_{ij} assign $x = op_{i|op_i|} \vee x = res_{|res|}$ then $\sigma_{ij} \neq 0$
- there are no additional constraint on repeated letters or columns

Input and Results

The solver that has been developed enumerate all the possible solutions and give them in the output with the counting. The input must be inserted with **blank spaces** between each letter. If *Count-sol = 0* is the only output then there's no solution for the problem, **note that** *Count-sol = 0* is printed by default.

Structure of the Solution:

General Idea:

The solution retrieval begin with the counting of columns, it has been decided to count from left to right, in parallel with the counting a facts that represent the problem of satisfy the summation of a column is asserted. After this phase the assignment to the variable starts from right to left, the direction has been decided because:

- first and last column has common constraint on the carryover: first column cannot use carryover to his addition and last column must not generate any carryover
- last column cannot contain zero elements (each row cannot contain 0 value for its last item)
- in case $|res| = |op1| + 1$ then the assignment to the last variable of res is forced to 1.

By this observation its reasonable to start from the end of the addition in the way to get more constraints and reduce the possible assignments from the beginning of the computation.
The first column from the left side is processed and some *crypto*-s are generated: a *crypto* is a temporaneous set of assignments which contains the assignment to each variable analyzed and a index k that represent last column analyze. Those *crypto*-s will *sweep* the addition from left to right and collect those assignments, generated by the arithmetic of the column, that don't contradict with the ones already found, the crypto can grow by the number of variables that differ from two subsequent column for each step. When $k = 1$ then the *crypto* has reached the end of the addition (which is the first column), a solution has been found and will be printed in the output.

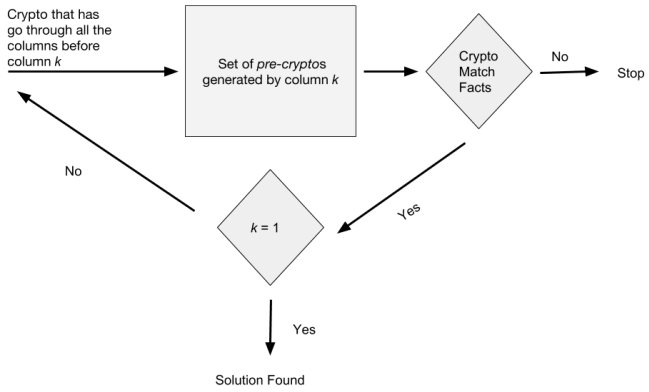

Figure 1: Flow of the crypto through the columns

Facts and Representation:

Representation of the problem:

- A def template has been used to represent the problem, the operands and the results, this will be got by input from the user:
```
(deftemplate add
(multislot op1)
(multislot op2)
(multislot res)
(multislot count))
```

- The digit will be defined in the way to enumerate the possible assignments, count-sol to keep trace of the number of solution found :
```
(deffacts init
(digit 0 1 2 3 4 5 6 7 8 9)
(count-sol 0))
```

Representation for computation:

- fact for enumeration to represent a possible assigment, this will be generated at the beginning:
```
(enum ?op ?digit)
```
- *pre-process* is the fact generated while counting columns; one assigned to each of them to trigger the computation, in this version *pre-process* is supported by *process* which is redundant as well with the rule **trigger**, the implementation works also without, but is kept to be used for a future further optimization as right now they have no relevant impact on the performance, this will be discussed later:
```
(pre-process ?x ?y ?z ?k)
```
x, y, z are the variables assigned to each character of the column and k is the index of the column
- *crypto* is the fact that *sweep* through the column to collect the compatible assignments generated by each column, the last column generates directly *crypto* if assignments to 1 has been enforced ($|res| = |op| + 1$)
```
(crypto ?l $?assignments ?i)
```
l is the index of the last column *swept* and i is the carryover required, the multifield *assignments* contains the assignments in this form $char_1\ d_1\ char_2\ d_2\ ...\ char_m\ d_m$
- *pre-crypto* is the fact that represent an arithmetic sum generated by the indexed column
```
(pre-crypto $?assignments ?k ?i ?j)
```
k is the index of column that generate this fact and i is the carryover generated, j is the carryover required, the multifield *assignments* contains the assignments in this form $char_1\ d_1\ char_2\ d_2\ char_3\ d_3$

Rules:

The most important rules that define the behavior of the systems are:

- **iterate:**
This rule has the role to go through every column from left to right, to count the number of column and generate the arithmetic problem for each column with the fact *pre-process*. This rule will follow the firing of **calculate** and will be deactivated after the firing of one of the variants of **end**. The fact *pre-process* has been connected to *process* (right now is redundant as already said) with the rules **trigger**.
```
(defrule iterate
(not(end))
?f1 <- (add (op1 $?a ?p)
(op2 $?b ?o)
(res $?c ?i)
(count ?l))
(pre-process ?x ?y ?z ?k)
(test(eq ?l (+ ?k 1)))
=>
(retract ?f1)
(assert (pre-process ?p ?o ?i ?l))
(assert (add (op1 $?a)
(op2 $?b)
(res $?c)
(count (+ ?l 1)))))
```
optimization:
not end has been put at the beginning and *add* template precede *pre_process* as add is always present as a single fact.
- **get_process** rule and his variants (there are variants for generated a unit of carry over and for compute the first column):
```
(defrule get_process
(tlen ?l)
(test(not (eq ?l ?k)))
...
(process ?op1 ?op2 ?res ?k)
constraint on the arithmetic validity
...
=> (assert (pre-crypto ?op1 ?d1 ?op2
?d2 ?res ?d3 ?k (div (+ ?d1 ?d2) 10)
?carryover)))
```
This is the rule that generated a arithmetic-valid *pre-crypto* to be *got* by a compatible *crypto*, $carry - over$

has to be defined by a specific variant of the rule.

optimization: where possible test has not been used, the equality relation between variables of has been verified using same variables symbol, first of all the facts that this is not the first column to be compute (as there's a specific variant for this case) in the way to not match useless facts

- **get_precrypto**:
```
(defrule get_pre_crypto
(crypto ?l $?stha ?i)
(pre-crypto ?op1 ?d1 ?op2 ?d2 ?res ?d3
?k ?i ?j)
...
constraint on the compatibility
between crypto and pre-crypto
...
(test (eq ?l (+ ?k 1)))
=>
(assert (crypto ?k $?stha ?op1 ?d1
?op2 ?d2 ?res ?d3 ?j)))
```
this is the rule that define the sweeping of the crypto among the column, this rules combine a crypto with a pre-crypto after the validity has been verified.

optimization: where possible test has not been used, the equality relation between variables of *crypto* and *pre-crypto* has been verified using same variables symbol, the matching of the fact *crypto* has been verified before as there are less matching facts than *pre-crypto*.
For this rule initially the *precrypto* was always assigned to the crypto and then the new generated fact was controlled and eventually *retracted*. Now variants of rule are used to optimize this phase and avoid redundant assignation, for this purpose the CLIPS function *subsetp* with the function *create* has been used in the way to distinguish number and letters.
```
(digit $?digits)
...
(test (not (subsetp (create$ ?opi)
(create$ $?digits))))
```

All the optimization of the rules as been approached as above, in the way to reduce the matched facts and to put the most constraint facts at the beginning of each rules. The prioritizing of the rules has been initially done by salience, but then the ordering was sufficient to accomplish the scope, the correctness of the execution of the rules has been tuned with the ordering with experiments but first of all design:

- The phases are ordered:
  - counting and generation of the problem (*pre-process*)
  - generation of arithmetic constraints for column k (*pre-crypto*)
  - generation of partial solution until column k (*crypto*)
  - print solution after *crypto* of column 1 has been asserted.
- The variants of **end** must have higher priority than **calculate** and **iterate**, as the ending column can be the first and if a column is the last must not be analyzed as a normal column with **iterate** rules, but with **end**. The firing of **end** will prevent the execution of the others;
- **first_round_get_process** has higher priority than other **get_process** for a little optimization for matching;
- **compact_crypto** rule must have higher priority than **get-pre_crypto** in the way to reduce the redundancy inside the facts before they are matched.

No module has been used.

Appendix:

Test cases:

Those are the test cases used for debuggin, to improve and verify the solution.

- `A + B = C` *32 solutions*
  `A + B = CD` *30 solutions*
  to verify particular case of single column
- `ABA + BCB = CDC` *13 solutions*
  `AAAA + BBBB = CCCC` *32 solutions*
  `AA + BB = CC` *32 solution*
  `ABA + BCB = ECDC` *11 solutions*
  to verify repeated column and repeated letter
- `AB + CD = EF` *476 solutions*
  to verify the number of solution (this problem has 476 solutions, this is an example several test have been done on this verification)
- `SEND + MORE = MONEY` *1 solution*
  base case of test after every change
- `SATURN + URANUS = PLANETS` *1 solution*
  `KYOTO + OSAKA = TOKYO` *1 solution*

  those and other test to verify performance in worst case, those were found on the web.
- `AB + BA = EB` *no solutions* `TOKYO + OSAKA = KYOTO` *no solutions*
  to verify no solutions case.

Results: All the reported test cases and other test conducted have given positive results, no bug has been found during final test phase. All the results has been checked with online solver: *Cryptarithmetic Puzzle Solver* (n.d.)
The performance are overall good, excluded some case in which it was required some seconds to find the solutions, the can be further reduced with the proposed optimization.

Future Development:

- **Reduce Constraints:**
  A reduction of the constraints has been already designed for different length of the operator: *pre-process* facts can be defined with different length instead of fixed length of 3 in which the last elements is always the results. A new definition of the rules **end**, **iterate**, **calculate** and **get-preprocess** is required with elastic matching for the fields: using multifield variable or variants of same rule.

- **Optimization:**
  A further optimization can be done and the code was thought from the beginning to lodge the following improvement.
  Right now a precise ordering of execution between the matching *crypto* and the generating of *pre-crypto* is not implemented, to be more specific right now the execution of the generation of *pre-crypto*-s for a column does not take place only *after* all the *crypto* of the previous column has been eventually matched, in the way to achieve this the usage of `(declare(salience x))` is required. The idea is to use dynamic definition of salience so that during *declaration, activation* and *every cycle of execution* salience for each rules will be defined. This function is allowed by **CLIPS** but it allows only to use *global variables* to define salience instead of *facts variables*, so this slowed the development of the initial idea, an implementation has been done but it's not ready to solve all the possible case. The rule **trigger** was designed initially to develop this idea.
  Why is this passage crucial?:
  The ordering of this execution phases is crucial because it allows to *retract* facts that will cause some redundancy in the generation: in the moment is sure that all the *crypto* until a column are defined, then is possible to retract all the enumeration (*enum*) that are not considered in this crypto. This action will dramatically reduce the number of *pre-crypto* generated by the subsequent columns and so the matching facts phases.

Why Rule-Based?

I want to express my opinion on the suitability and the power of the rule-based approach to this kind of problem, this opinion has been developed after the work on this project as i had no previous experience on developing a rule-based solution, some paper not related to this particular problem has helped me to understand why rule based is so important and to which problem it can be applied.
First of all this kind of approach was really open-minding as it was completely new for me and has helped me to develop a new understanding of a problem and expand the way to approach any problem in general. Rule-based has a strong impact in the way you represent the problem, at first the language seems less powerful than an imperative one but it has different kind of instrument to attack a problem. I think rule-based is really strong when the problem has recurrent pattern and can be reduced to smaller similar problems that have a clear structure and are subject to evident rules. By this statement is easy to see why rule-based can be a good choice to attack a crypto-arythmetic problem. This kind of puzzle has recurrent pattern and their *small problem* are the column summation to which you can reduce the problem to solve it from the smaller and then find relations between solutions between each small problem. In the imperative paradigm this kind of reduction is possible, but is not as natural as in a rule-based system, you can build up function and entity with the object-oriented paradigm in the way to recreate this kind of pattern but the representation require higher overhead and is more *verbose*. Considering *abstraction and representation* the rule-based approach is more adherent because the concrete execution is abstracted by the *inference engine* from which the actual performance depends, the concrete reasoning on the semantic of the rules and the matching of the facts define the performance trade-off between the imperative and rule-based paradigm.

Credits:

For this project I used as main reference Wygant (1989) and Riley (n.d.) to have an overview of CLIPS enviroment and solve doubt about strategy of firing rules. To have easy-access reference for clips i've used *Guide to Clips* (n.d.) and *Cryptarithmetic Puzzle Solver* (n.d.) to check the solution

References

1. *Cryptarithmetic Puzzle Solver* (n.d.), `http://bach.istc.kobe-u.ac.jp/llp/crypt.html`.
2. *Guide to Clips* (n.d.), `https://www.csie.ntu.edu.tw/~sylee/courses/clips/bpg/top.html`.
3. Riley, J. C. G. G. (n.d.), Expert systems: Principles and programming, *in* C. Publishing, ed., 'Elementary particle theory'.
4. Wygant, R. M. (1989), 'Clips—a powerful development and delivery expert system tool', *Computers & industrial engineering* **17**(1-4), 546–549.

Lorenzo Ciampiconi
A0179901X
E-mail: e0271056@u.nus.edu