Solving Cryptarithmetic puzzles with CLIPS

Marius Urbonas
School of Computing
National University of Singapore
Marius.urbonas.edu@gmail.com

Abstract

This paper covers an approach taken in order to solve cryptarithmetic addition problems along sided possible implementation decisions with that can be made whilst programing this in CLIPS language for this and general Constrain Satisfaction Problems.

1 Introduction

Cryptarithmetics problems usually expressed as two or more strings being the operands in addition and one string which represents the sum. Note that each unique letter should correspond to a distinct digit from 0 to 9, while adhering to the constraints imposed by long addition, for example:

```
  S E N D
+ M O R E
---------
M O N E Y
```

Cryptarithmetic problems belong to the larger scope of Constraint Satisfaction Problem (CSP) set which is proven to be NP-complete in general [Mackworth 1977] and therefore is no known polynomial time algorithm exists to solve this problem. There are various ways one can approach solving CSP problems depending on what kind of a solution is required, but in this paper I focus on finding all possible solutions for a given cryptarithmetic problem of exactly two strings representing operands and one string representing the sum.

There exists various approaches that can be used to tackle CSPs. They include integer programming techniques (cutting plane methods and branch and bound), this can be used to find an exact solution. On the other hand, there are various approaches that provide an approximate solution, including local search methods (simulated annealing, threshold accepting, tabu search and genetic algorithms) and neural networks. However, technique that is widely used for exploring all of the solution space for a particular CSPs is tree search combined with backtracking and consistency checking.

2 Approach details

Solving the cryptarithmetic puzzle given in the introduction in a simple very brute-force approach would be to try to enumerate all possible assignments to the equation:

$$D + 10N + 10R + 101E + 100O + 1000S + 1000M = Y + 10E + 100N + 1000O + 10000M$$

This does not seem very feasible even for this problem, because it has only one constraint involving all eight variables, which is not of much use for constraint propagation. The problem can be reformulated by adding extra variables $C_1, C_2, \dots C_4$ each with domain $C_i \in \{0,1\}$, representing the quantities carried from long addition, now the constraint representing the sum can then be rewritten as:

$$D + E = 10 * C_1 + Y$$
$$N + R + C_1 = 10 * C_2 + E$$
$$E + O + C_2 = 10 * C_3 + N$$
$$S + M + C_3 = 10 * C_4 + O$$
$$C_4 = M$$

$$In\ general:$$
$$Op1_i + Op2_i + C_{i-1} = Ans_i + 10 * C_i$$

Where $Op1_i$ is ith variable in the first operand, etc.

This splits the original problem into sub-problem system. Here, I will explore the possibility of assigning the values to all variables of a specific column in one step, by assigning such 3-tuple $(n_{1i}, n_{2i}, n_{1i} + n_{2i} + c_{i-1})$ to $(Op1_i, Op2_i, Ans_i)$ where $n_1, n_2 \in \{0,1 \dots base\}$, (base being the number base used in the problem, usually decimal) and these values satisfy the constrains of the problem. Having a fixed a certain carry vector assignment $\bar{C} = < C_1, C_2, \dots C_4, >$ limits the number of assignments that can satisfy one column with three unassigned variables - $num_{3-tuples} \leq 32$, where in the decimal case which is quite a big branching factor on its own. But looking more closely it can be seen that this would only be the case for the first assignment in the worst case. Having at least one variable assigned in a column given that 3 variables are already assigned would reduce number of assignments to $num_{3-tuples} \leq 6$. Furthermore, if two variables are assigned a value for a given column, then only one possible 3-tuple exists that could satisfy that column given a fixed $\bar{C}$. This shows that even though branching factor at first is quite large, it can be significantly reduced going down the tree if an appropriate heuristic is chosen. In this approach max-constrain heuristic is chosen to be used, that is to a choice is made to prioritize assigning values to a column

that has the most variable appearances in other columns. In that way it is tried to force conflicts as early as possible. This helps, because all of the assignment space is being explored to gather all of the solutions, hence terminating unsatisfactory branches as early as possible will improve the efficiency of the solution. Of course this means that to to explore the full assignment space you have to do this for all possible permutations of a carry vector $\bar{C}$.

3 Implementation overview

Implementation of the previously described approach is presented here, most important modules used to control the workflow are covered in the following sections, here is a quick overview:

SETUP module is used to preprocess the given data, create the required facts from the input as well as infer some additional constraints from the particular problem structure therefore limiting the search space before the search begins.

MAIN module handles module execution cycle as well as terminates the program when solution space was searched completely.

BRANCH module which using a max-constraint heuristic choosing a next column to branch along as well as finds the value assignments for particular column.

PROPAGATE module which uses the assignments to infer new values for unassigned variables and finds a conflict with any constraints which triggers a backtrack.

3.1 SETUP module

At this stage most of the required facts for program execution are asserted.

This module uses some insights about the problem to further reduce the space. For example, given a column of the form:

$$X + Y + C_{i-1} = X + 10 * C_i$$
or
$$Y + X + C_{i-1} = X + 10 * C_i$$

It can already be inferred that Y is assigned $0 \oplus 9$, or if a given base system is not decimal $0 \oplus (base - 1)$. A rule matching such patters is used to set zero-nine flag to TRUE in variable Y. This combined with the flag called nonzero set for the first letter of each string may allow to infer a value before the search has even started therefore reducing the search space.

To take advantage of the fast pattern matching, a design decision is to precompute all of the possible assignments to columns as facts of 3-touples using the form:

$$(\text{triplet } (n_1, n_2, n_1 + n_2 + c))$$

Because there are two variables that can be freely chosen from domain of size $base$ and c is chosen from domain of size two, the whole number of facts pre-generated equal to $base * base * 2$, hence for base 10 it would be 200 facts of possible assignment for column values. This is not too big for modern computers, but additional steps have to be used to make sure the program generates as little partial matches for column assignments with rules as possible to make it efficient.

3.2 MAIN module

This module holds almost all of the templates used in this program, which gives it nice control and consistency whilst switching between modules as well as allows for other modules to just focus on required rule productions.

The most important templates defined here are var and $current\ stage$. The var template is of the form:

$$(var(\ lit)(value)(\ stage)(nonzero)(zero\ nine))$$

Where lit holds the letter associated, value slot holds the value assigned at the current branching stage, stage is the current depth of a search tree, nonzero and zero-nine are flags that are set during the SETUP module to limit the scope of the variable.

Another useful template is defined in a form of:

$$(current\ stage\ (stage)\ (localframe)\ (frame)(current\ branch\ choise))$$

Here the important bits are the $local\ frame$ which takes note of what triplet facts had already been tried whilst branching in this stage to avoid repeating the same assignment, as well as $frame$ which keeps the ids of the columns that have already been used to branch along up until this stage.

A set of simple rules in this module is used to check whether the solving process should be stopped, the solution is found or the carry vector $\bar{C}$ need to be updated to a new permutation. The rule driving the search process is called solve and it is used to update the $current\ stage$ variable to next value as well as push new modules onto the focus stack to go through a cycle of BRANCH, PROPAGATE modules before returning control to the main module.

3.3 BRANCH module

Here the heuristic value is computed at each branching stage and the appropriate column which constrains the most other columns is chosen to branch along.

All rules in this module reuse first chunk of the pattern matching node as they all start the same pattern which tries to reduce the strain on the matching process The rule $brach\ triplet$ chooses appropriate column values, makes the assignments, and pops the focus stack.

Patterns in this rule are placed in an order which minimizes the number of matches for the patters following, therefore limiting the number of partial matches. Very first pattern match picks the current stage fact with the highest value of stage which has to use an expensive forall method to make sure that no bigger stage exists, but once it is chosen it forces only 1 possible value for each of the following patterns except for the possible assignment 3-touple which can produce at worst 45 different matches which, as described in the previous section, can then be reduced to 32 activations in the worst case. The 45 different possible matches are not that bad considering that every pattern before can only have one fact match hence it is not computationally horrible.

This rule has higher salience compared to the rule $no\ branches\ left$ which allows it to be fired if there are no new possible assignments and which, in turn, forces a backtrack.

3.4 PROPAGATE module

Propagation module does a lot of the heavy lifting finding columns for which enough variables are known to infer the last one unassigned in that column.

There are 4 different propagation rules, where each has equal patterns at the front to reuse the pattern matching nodes, these rules cover all of the possible situations where it is possible to infer a new value, these are presented below:

1. $X_i + Y_i + C_{i-1} = Z_i + 10 * C_i$
2. $X_i + Z_i + C_{i-1} = Y_i + 10 * C_i$
3. $Z_i + Y_i + C_{i-1} = X_i + 10 * C_i$
4. $X_i + X_i + C_{i-1} = Z_i + 10 * C_i$

Here X_i and Y_i are assumed to be already assigned a value and Z_i is the value to be inferred. As there $\bar{C}$ value is fixed there will always be only one such assignment for Z_i, which if the checked against the constraints is inconsistent, a backtrack is initiated, otherwise the control is returned to the MAIN module to continue the solve cycle.

Patterns for these rules are again put in such a way, which tries to minimize partial matches. The initial $current\ stage$ fact match which requires not predicate to find the most up to date $current\ stage$ might be expensive although in practice there tends to be less than 6 such facts at the time. Columns pattern is matched first, as usually for a bigger problem there is smaller number of columns than there are distinct letters, and a choice of column forces only one possible match for corresponding variables, which in tern limits the number of possible triplet facts that can match to exactly one.

3.5 Additional notes

The program does not take effort to validate that the input is in a correct format, therefore it is left for the user to make sure that the input follows basic cryptarithmetic problem rules.

Program allows to enter number base, but for these test particular examples it is assumed that the base is decimal.

Following inputs described in the appendix are all possible cryptarithmetic puzzles with at least one solution and the program correctly solves them all.

4 Discussion

Rule-based implementation so solve CSP problems is clearly possible and offer readable and uncluttered code, but does not seem to be as efficient as say imperative implementation of the same algorithm. The constraints are not used actively whilst searching for the solution and there are particular difficulties expressing some constraints as the language does not support native way to express constraints such as disjunction constrains for example exclusive or for a set of variables.

References

[Mackworth 1977] Alan K. Mackworth. *Consistency in network relations.* Department of Computer Science, University of British Columbia, Vancouver, Canada 1977.

Appendix

$$SEND + MORE = MONEY$$
$$DONALD + GERALD = ROBERT$$
$$BASE + BALL = GAMES$$
$$LOGIC + LOGIC = PROLOG$$
$$LINEAR + LOGIC = PROLOG$$