Skeletal Division Satisfiability Solver

[redacted]

CS2800: Logic and Computation

Prof. Jason Hemann

November 9, 2020

Abstract

This paper focuses on the encoding of a SAT solver for cryptarithmetic skeletal division puzzles. We show how the solution to any given skeletal division puzzle can be represented by a conjunction of propositional logic constraints. By considering a vast sample of skeletal division puzzles, we write a program implementing a Z3 SAT solver in Python. We demonstrate how arithmetic and domain constraints can be generalized for all possible problems. Ultimately, we produce a satisfiability solver capable of solving any skeletal division puzzle if adequate time is allotted to it.

Skeletal Division Satisfiability Solver

[redacted]

Contents

General Problem Area

Background

Verbal Arithmetic (aka. Cryptarithmetic) puzzles are types of math games consisting of a mathematical equation whose digits are represented by letters. These types of puzzles frequently appear in recreational math books and are often used as exercises for teaching algebra in an introductory capacity. There are a few variations on this puzzle, but the most common type is alphametic; in an alphametic puzzle, there is a set of words with a simple mathematical operation applied to them, usually addition. Each unique letter represents a decimal digit and there are no leading zeros. The game dates to the late 19th century [2], but the most common example, *Figure 1*, dates to 1924.

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

Figure 1: Oldest examples of a cryptarithm alphametic puzzle

The type of cryptarithmetic puzzle represented by *Figure 1* has often been the study of computer scientists since the puzzles provide a good example of brute force and backtracking algorithms. Alphametic cryptarithms have also been studied within SAT since they can be modeled as constraint satisfaction problems. Using modulo arithmetic, the puzzle can be written as a series of equations using the sums in each digit place [1]. When the puzzle is solved in base 10, the number of possible solutions is limited to 10! / (10-L)! for L distinct letters less than or equal to 10. This limit holds under the assumption that no two distinct letters represent the same integer. However, in his paper on the NP-completeness of Cryptarithms, David Eppstein found

that when the puzzle is expanded to be in an arbitrary base determined by the constraints of the equation, the problem quickly becomes NP-complete [3].

Rather than focusing on alphametic cryptarithms, this paper focuses on another, less popular, variation of Cryptarithmetic puzzles called skeletal division. Instead of posing the puzzle as a set of words with a mathematical operation applied, each letter in the puzzle represents a digit in a long division problem. Unlike alphametic cryptarithms, in skeletal division unique letters are permitted to equal the same value. However, each letter can only correspond to a single digit. In other words, every instance of a given letter must be equal. Consider the division puzzle in *Figure 2*.

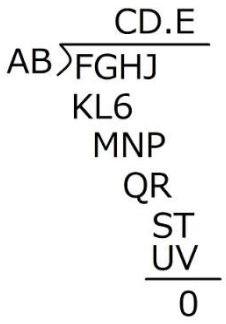

Figure 2: Original skeletal division example

Notice that every letter of the puzzle in *Figure 2* is unique, but multiple letters can correspond to the same digit. For instance, the solution to the example (*ABCDEFGHJKLMNPQRSTUV* = 14975136512105987070), has the letters *A*, *F*, *K*, and *M* equal to 1.

Since distinct letters do not necessarily have to correspond to unique digits, the letters are often treated as place holders rather than variables. Furthermore, in some division puzzles the

letters are replaced with a single symbol to avoid possible confusion surrounding repeated digits. A popular example of such a puzzle is Feynman's Problem given by *Figure 3*.

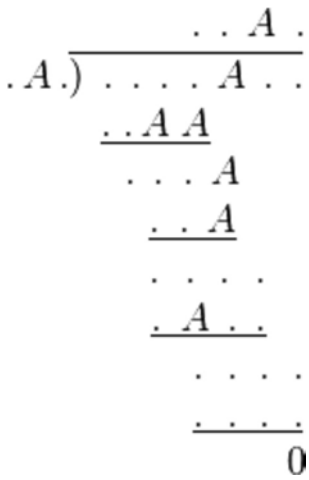

Figure 3: Feynman's Division Problem [8]

As seen in *Figure 3*, Feynman's Problem replaces letter variables with periods. The only letter not replaced was *A*, to indicate that the numbers at the locations of *A* were equivalent.

Approach

Verbal Arithmetic is classified as a constraint SAT problem (CSP). In CSPs there is a set of variables, X_i, each with a respective domain of values, D_i, and a set of constraints C on the variables [4]. In an alphametic cryptarithm, each letter in the puzzle is a variable, with its own respective domain of 0-9 (or 1-9 if it is a leading digit) and a set of constraints based on the modulo mathematical operations being applied to the digits.

Similar Problems

Other more well-known CSPs include the eight queens puzzle, the map coloring problem, and sudoku with its multiple variants. The general approach to CSPs is to use constraint

programming to encode the problem into the set of variables, domains, and constraints, then pass those into a combination of SAT and SMT solvers. The oversimplification of what the solver does is loop through propagating values then guessing and backtracking if needed. When the solution for a variable can be found using the constraint expressions and previously assigned variables, the solver propagates that value and continues to check if any more can be found. When the solver reaches a point where the next variable is unknown, it guesses for that value and repeats attempting to propagate values. If an expression is reached where all the variables are assigned and the Boolean expression is false, the solver backtracks to its last guess, changes it, then repeats the process. The propagation of values is based on the heuristics of the problem being solved. In the example of sudoku, the constraints would be that there are no repeated numbers in a box, row, or column. Therefore, a heuristic to propagate values in a sudoku board would be that if a box, row, or column has only one blank, the blank must be the one non repeated digit.

Utilizing Modern SAT and SMT Solvers

Since SAT solvers take a set of clauses in CNF, modeling arithmetic purely in SAT adds a second layer of difficulty to the encoding of the skeletal division problems. Therefore, using a modern SMT solver significantly simplifies the encoding process for skeletal division problems. SMT solvers provide theories about arithmetic with bitwise operations, allowing the encoding to take unbounded integers, real numbers, bit vectors, arrays and more [7].

In this paper, the source code element utilizes z3, an open source SMT solver released by Microsoft Research. Z3 is available in a variety of interfaces including C++, Java, and Python. The source code is written using a Python wrapper for z3. By utilizing an SMT solver rather than

a purely SAT solver, the encoding for CSP problems can be written as a CNF of constraints on natural numbers rather than having to encode the bitwise operation of Boolean values ourselves.

Contributions

Given the general roadmap previously presented, our contributions are as follows. We present a novel encoding for the constraints of a lesser known variant of a previously studied game. Taking into account the encoding of alphametic cryptarithms: we identify the unique requirements of skeletal division puzzles, find an encoding to the constraints of all variables, and utilize a modern SMT solver which takes the encoding and determines whether the puzzle is satisfiability and returns the solutions if any exist.

Metric for Success

The metrics for success are based on whether the source code can take the example skeletal division provided in *Figure 2* and correctly identify the single valid solution.

Methodology

In order to encode the problem of skeletal division, it is important to understand the encoding of alphametic cryptarithms. In alphametic cryptarithms, the variables in the expression must first be identified, then domain of variables specified, and finally, constraints about the problem must be written. The first two are straight forward. When given a cryptarithm, the variables are the letters in the equation and their domains are the decimal digits (0-9). The constraints on a cryptarithm can be found using the rule that there are no leading zeros and the modulo arithmetic of the puzzle. In the example from *Figure 1*, *SEND* + *MORE* = *MONEY*, since there are no leading 0's, the domain of *M* and *S* are reduced to the digits 1-9. Then, using modulo arithmetic, constraints for each place value in the addition problem can be written, i.e. *D*

+ E = Y % 10, N + R = E % 10, etc. and the final constraint is whether the original addition problem also holds.

Encoding the Problem

The process to encode skeletal division cryptarithms follows similar steps as alphametic cryptarithms but, given that the puzzle is not a simple addition problem like in alphametic cryptarithms, the arithmetic constraints are drastically different. Ultimately, the skeletal division problem can be constructed with a combination of domain and arithmetic constraints, but first, the data of the inputted puzzle must be organized into usable data types. The elements of the inputted string are added to a dictionary, *words*, along with each element of the inputted array. From the elements in *words*, *letters* is defined. *Letters* is a dictionary of every distinct letter present in the elements of *words*. For each value in *words* and *letters* the associated key is an integer variable in Z3 of the same name as the value. In the case that the puzzle includes predefined numbers, the key of each such number is set to the integer value of the number rather than creating a new integer variable in Z3. In doing so, the predefined numbers are kept constant throughout the solving processes. In contrast, the values of the integer variables are adjusted by the solver with respect to the domain constraints.

Domain Constraints

Domain constraints dictate every variable's range of possible values. Our program contains three vital domain constraints: *lettersToNum*, *wordsToNum, wordNotZero*. The encoded definition for *lettersToNum* is as follows,

```
lettersToNum = [ And(0.0 <= v, v <= 9.0) for l,v in letters.items() ]
```

The code above applies a constraint on the range of values a letter's key can have. The *lettersToNum* constraint restricts a letter's value to an integer between zero and nine, inclusively.

For the example in *Figure 2,* the series of propositional logic statements that make up the *lettersToNum* constraint is as follows:

```
[And(F >= 0, F <= 9),
 And(G >= 0, G <= 9),
 And(H >= 0, H <= 9),
 And(J >= 0, J <= 9),
 And(A >= 0, A <= 9),
 And(B >= 0, B <= 9),
 And(C >= 0, C <= 9),
 And(D >= 0, D <= 9),
 And(E >= 0, E <= 9),
 And(K >= 0, K <= 9),
 And(L >= 0, L <= 9),
 And(True, True),
 And(M >= 0, M <= 9),
 And(N >= 0, N <= 9),
 And(P >= 0, P <= 9),
 And(Q >= 0, Q <= 9),
 And(R >= 0, R <= 9),
 And(S >= 0, S <= 9),
 And(T >= 0, T <= 9),
 And(U >= 0, U <= 9),
 And(V >= 0, V <= 9),
 And(True, True)]
```

These propositional statements define the range of each unknown letter, *A – V,* and verify that the predefined numbers, 0 and 6, also follow the constraint. Since all the domain constraints of the variables are in CNF, if even one of the variables is set to a value outside its specified domain, the entire clause returns false. This means a possible solution to the puzzle cannot hold unless all the values for the variables are within their specified range.

The second constraint, *wordsToNum*, defines the domain of a word's value.

```
wordsToNum = [ v == Sum(*[letter_symbol * 10.0**index
                for index,letter_symbol in enumerate(
                    reversed([letters[l] for l in list(w)]))])
                                for w, v in words.items()]
```

The code for *wordsToNum* states that the value, or key, *v*, of each *word* in *words* must be mathematically equivalent to the integer parsing of the corresponding integer variable. Once again considering the example in *Figure 2*, the logic statements for *wordsToNum* are

```
FGHJ == J*1 + H*10 + G*100 + F*1000,
AB == B*1 + A*10,
CD == D*1 + C*10,
E == E*1,
KL6 == 6 + L*10 + K*100,
MNP == P*1 + N*10 + M*100,
QR == R*1 + Q*10,
ST == T*1 + S*10,
UV == V*1 + U*10,
0 == 0
```

The logical equivalence statements above demonstrate the application of base-10 convergence to determine the necessary value of a given word. Additionally, as with *lettersToNum*, every statement must hold true or the entire statement fails.

Lastly, the constraint *wordNotZero* contributes to the word value domain with the following code.

```
wordNotZero = [ Or(And(Or(num == arr[len(arr)-1], num == remain), len(num) == 1),
                                letters[num[0]] != 0) for num in words.keys()]
```

The constraint *wordNotZero* prevents all words from being set to a value with a leading zero unless the word is the last element of the inputted array. The last word in the array of intermediate can equal 0 because in a long division problem, if there is no remainder, the final step will result in 0. The propositional logic statement present in the code for *wordNotZero* is more complex then prior statements because of the nested operations. As such, it is easier to understand by analyzing the equivalent statements for the example in *Figure 2*, which are,

```
Or(And(Or(False, False), False), F != 0),
Or(And(Or(False, False), False), A != 0),
Or(And(Or(False, False), False), C != 0),
Or(And(Or(False, True), True), E != 0),
Or(And(Or(False, False), False), K != 0),
Or(And(Or(False, False), False), M != 0),
Or(And(Or(False, False), False), Q != 0),
```

```
Or(And(Or(False, False), False), S != 0),
Or(And(Or(False, False), False), U != 0),
Or(And(Or(True, False), True), False)
```

The example above shows that *wordNotZero* sets restrictions on the first letter of a word which in turn restricts the possible values of the word. Also, by considering the outermost OR, if the first letter of the word does not equal zero, the statement is evaluated to true. On the other hand, if the letter does equal zero, it must fit under the accepted list of exceptions for the statement to evaluate as true. The exception cases are defined by the nested AND and OR operations. In English, the letter must have a length of one and either be the remainder or the last value in the array. This constraint along with *lettersToNum* and *wordsToNum* successfully control the possible values of the integer variables in the Z3 solver.

Arithmetic Constraints

Arithmetic constraints define the initial puzzle as an algebraically equivalent series of sound alphametic cryptarithms. For skeletal division, each puzzle has at least five unique equational rules. These rules are formed by several sub-equations. The collection of sub-equations is stored in the list *equations* which is ultimately used to define the constraint:

```
checkEquations = [ (eval(equ, None, complete) == True) for equ in equations ]
```

The *checkEquations* constraint contains the individual equations that must hold true for the current problem. *checkEquations* in it of itself is conjunction of the equations in *equations* where the evaluation of each equation must be true.

The first category of equations checked by *checkEquations* place constrains on the value of the remainder. If a remainder is present (a decimal is in the quotient) then, the value of the word following the decimal must equal 10 to the power of the length of the word times the mod of the dividend and divisor divided by the divisor. For example, the remainder equation for the

example in *Figure 2* would be `E == (10*(FGHJ%AB))/AB)`. In the case where a decimal is not present in the quotient and thus there is no remainder, three different equations are defined. The first states that the last element in the array must equal zero because if it were any value other than zero, then the divisor would not go into the dividend evenly and there must be a remainder. The second equation simply states that since there is no remainder, the mod of the dividend and divisor must equal zero. The last equation verifies the soundness of the initial division problem by checking an equivalent multiplication problem: divisor * quotient = dividend.

The purpose of the second category of equations checked by *checkEquations* is to verify the validity of the overall division problem. Simply, the one equation states that the dividend divided by the divisor must equal the quotient. If a remainder is present, it removes the remainder from both sides of the equation.

The objective of the next set of equations is to dictate that the difference between the divisor and first member of the array must equal the second member of the array. For *Figure2* the equation for this constraint is `(FGHJ - FGHJ%1)/1 - KL6*10 == MNP`. Notice that the equation is not as simple as the equivalent step in long division is. The members of the array needed to be adequately shifted along with the dividend. The process of determining a general equation for the proper shift in terms of given data resolved to be very difficult. In fact, defining a general equation to define each subtraction step in the long division proved particularly hard. In the end, there was no single constant equation that could be uniformly applied to every possible input. Instead, the following equations was written with the variables *shift* and *powerOfTen* adjusted depending on the characteristics of the given puzzle.

```
    equDiffBase = '((' + dividen + '-(' + dividen + '%pow(10,' + str(powOfTen) +
'))))/pow(10,'  + str(powOfTen) + '))-(' + arr[0] + '*' + str(shift) + ')==' + arr[1]
```

The *powerOfTen* in the equation above was determined by the difference between the length of the first element in the array and the length of the divisor. The *shift* was determined by the difference between the length of the dividend and the length of the quotient. Fortunately, this equation is only used for the base case, dividend minus first element in the array. For the subsequent cases of subtraction, the element in the array which represents the expected difference is shortened to the necessary values rather than shifting the subtrahend and minuend. To better understand the purpose of these equations, examine the example from *Figure 2*:

```
(MNP - QR == S),
(ST - UV == 0)
```

Notice from the equations that instead of applying modular arithmetic to shift the values of the subtrahend and minuend, the difference is truncated to the necessary length.

The final group of equations contained within the *checkEquations* constraint aim to structure the relationship between the digits of the quotient and the divisor. Generally, the product of a single digit in the quotient and the divisor must equal the corresponding long division step. Therefore, in terms of *Figure 2*,

```
(C*AB == KL6),
(D*AB == QR),
(E*AB == UV)
```

The equations of the form above are relatively straight-forward in terms of their encoding and derivation. With all the equations defined, the *checkEquations* constraint is asserted in the solver along with the three domain constraints.

Results

As stated previously, our metric for success is based on whether the source code can take the example skeletal division provided in *Figure 2* and correctly identify the single valid solution. Consequently, *Figure 4* below displays the output of our source code when *Figure 2* was inputted.

```
FGHJ / AB == CD.E
{'FGHJ': 1365, 'AB': 14, 'CD': 97, 'E': 5, 'KL6': 126, 'MNP': 105, 'QR': 98, 'ST': 70, 'UV': 70, '0': 0}
*** 1 solutions found in 47.3s ***
```

Figure 4: Source code output for the input from *Figure 2*

Figure 4 clearly shows that our source code successfully produces the solution. Additionally, the program recognizes that there are no other valid solutions. From the results in *Figure 4* along with numerous subsequent tests, it can be concluded that the logic built into our code is effective in computing a skeletal division problem. Given these results, our project was a success.

Summary

In conclusion, we were able to successfully encode the constraints of a skeletal division problem. By examining the lesser known variant of cryptarithmic puzzles, we provided a novel encoding to a problem not previously solved using Boolean SAT and SMT. Because skeletal arithmetic is a constraint satisfaction problem, identifying the constraints of a skeletal division allows the reader to gain a better understanding of constraint programming, which can be used to tackle more difficult problems within SAT. There still remain open problems within SAT which are posed as a series of decision problem. By understanding a simple, novel puzzle, this

understanding can hopefully be transferred forward into more difficult and non-trivial problems in SAT.

References

[1] Cryptarithmetic Puzzles | OR-Tools | Google Developers. (n.d.). Retrieved November 10, 2020, from https://developers.google.com/optimization/cp/cryptarithmetic

[2] The American Agriculturalist. (2011). Retrieved November 10, 2020, from https://archive.org/stream/americanagricult23unse

[3] Eppstein, D. (2000, June 8). On the NP-Completeness of Cryptarithms. Retrieved 2020, from https://www.ics.uci.edu/~eppstein/pubs/Epp-SN-87.pdf

[4] Stuart Jonathan Russell; Peter Norvig (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall. p. Chapter 6. ISBN 9780136042594.

[5] Smock, J. (Writer). (2016). *A Peek Inside SAT Solvers* [Video file]. Retrieved 2020, from https://www.youtube.com/watch?v=d76e4hV1iJY&ab_channel=ClojureTV

[6] Baseel, C. (2020, February 14). Can you solve this crazy difficult, super satisfying math puzzle from a Japanese middle schooler? Retrieved November 10, 2020, from https://soranews24.com/2020/02/13/can-you-solve-this-crazy-difficult-super-satisfying-math-puzzle-from-a-japanese-middle-schooler/

[7] Compose Conference (Producer). (2016). *Analyzing Programs with Z3* [Video file]. Retrieved 2020, from https://www.youtube.com/watch?v=ruNFcH-KibY&ab_channel=ComposeConference

[8] Feynman's Division Problem. (2014, August 16). Retrieved November 10, 2020, from https://beyondmathsolutions.wordpress.com/2014/08/22/soln-feynmans-division-problem/