Cryptarithmetic Puzzle Solver Report

Zhe Lin 109369879

Introduction

A cryptarithmetic puzzle is a puzzle consisting of an arithmetic problem in which the digits have been replaced by letters of the alphabet. To solve the puzzle, we need to assign each letter in the puzzle an identical digit using the constraints provided by arithmetic. Figure 1 shows a well-known example published in the July 1924 issue of Strand Magazine by Henry Dudeney:

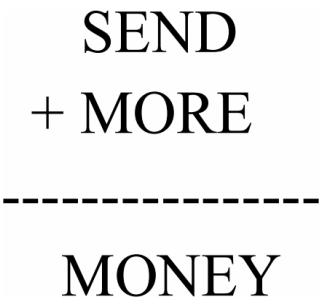

Figure 1: Cryptarithmetic puzzle example 1.

Assigning digits to letters in the following way would be a solution for the example in Figure 1:

O=0, M=1, Y=2, E=5, N=6, D=7, R=8 and S=9.

Constraint satisfaction problem is a main topic covered in our AI course this semester. A CSP can is a problem with a given set of variables, together with a finite set of possible values that can be assigned to each variable, and a list of constraints. The goal of such a problem is to find values of the variables that satisfy every constraint. Cryptarithmetic puzzle is a kind of CSP. In

this case, the set of variables are the letters and the set of values are the ten decimal digits. Besides, as mentioned by Abbasian and Mazloom, there are four constraints as shown below:

1. The arithmetic operations are in decimal; therefore, there must be maximum ten different letters in overall strings which are being used.
2. All of the same letters should be bound to a unique digit and no two different letters could be bounded to the same digit.
3. As the words will represent numbers, the first letter of them could not be assigned to zero.
4. The resulting numbers should satisfy the problem, meaning that the result of the two first numbers (operands) under the specified arithmetic operation (plus operator) should be the third number.

The motivation of implementing the cryptarithmetic puzzle solver is that I would like to apply what I learned from class and some research paper to solve a real-world fun problem. I implement two algorithms to solve this problem: depth first search with backtracking and genetic algorithm.

Implementation:

Constraint 1 and constraint 3 as mentioned above can be checked or fixed during the process of input reading. Constraint 2 can be satisfied when assigning a letter a digit by checking the assigned letters.

For constraint 4, we can expand it a little more. Figure 2 is another cryptarithmetic puzzle example:

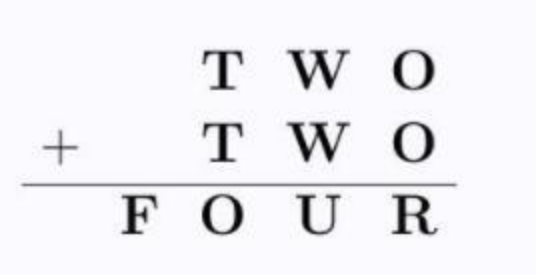

Figure 2: Cryptarithmetic puzzle example 2.

We can expand constraint 4 in such a way:

O+O=R*10+X1, X1+W+W=U*10+X2, X2+T+T=O+10*X3, X3=F

(Xi's are auxiliary variables with domain {0, 1})

DFSB

I did DFSB in two ways, one is plain DFSB, and the other is DFSB+ with variable ordering and AC3. For DFSB, I used the expanded constraint 4 as shown above. The DFSB and AC3 algorithms are modified from pseudocode from lecture slides.

For variable ordering, I try two ways. First, in mode 1, is to use the most constrained variable ordering heuristic. The variable that shows up in more constraints expanded from constraint 4 have higher priority. In mode 2, I choose the variable with smaller domain first. It turns out the second way works better. I will explain the difference of their performances in the next section.

Genetic Algorithm

I modified the algorithm in the paper by Abbasian and Mazloom. This algorithm has a sense of min-conflict algorithm as we implemented in homework assignment. The main idea is to first randomly assign digits to letters. We obtain such multiple such assignments as initialization. In each iteration step, a pair of digits are randomly chosen, and their assigned letters are swapped.

We compare the conflicts for the new assignments and the old assignment. And then we choose the identical assignments with smaller conflicts for next iteration step. We keep repeating such steps until we get an assignment with 0 conflict.

The way we calculate the conflict is as follow:

Using example, if we assign T=3, W=5, O=7, F=8 and U =6 and R=2, then,

conflict = abs(FOUR-(TWO+TWO))=abs(8762-(357+357))=8048.

In the paper, Abbasian and Mazloom use this algorithm in parallel. In my implementation, I just simply use loops.

Results and Analysis

I obtained the test data from the daily puzzles on “mathmisery.com”.

Usage of the two files:

python ga.py <input_file> <output_file>

python dfsb.py <input_file> <output_file> <mode_flag>

The following results could be different on different trials.

	DFSB	DFSB+ mode1	DFSB+ mode2	GA
Easy: q76	0.38s	1.86s	0.036s	0.006s
Easy: q73	8.45s	0.039s	2.35s	0.131s
Hard: q69	316.73s	659.356s	11.38s	0.095s
Hard: q79	352.24s	146.35s	3.67s	0.042s

The harder cases are those with longer operands' lengths. For the easy cases, the performances between the three modes for DFSB are not large. However, for the harder case, mode 2 performs much better than mode 1 and plain DFSB. We could see that exploring the variables with smaller domains first improve the DFSB's performance a lot. This is the main factor. For cryptarithmetic puzzles, there are at most 10 variables (excluding the auxiliary variables) and 10 values. Therefore, applying MCV heuristic and AC3 algorithm might not improve the performance a lot.

For the genetic algorithm, assignments are generated randomly, the performance is based on luck. But generally, it is much faster than DFSB, which is similar to that min-conflict algorithm performs better than DFSB in homework 2.

Future Work

My implementation only solves for the addition/subtraction problems with inputs in format of: FIR+/-SEC=RES. In future implementation, I should also solve the problems with more complex inputs with more operands and operators.

Reference

Abbasian, Reza, and Masoud Mazloom. "Solving Cryptarithmetic Problems Using Parallel Genetic Algorithm." *2009 Second International Conference on Computer and Electrical Engineering*, 2009, doi:10.1109/iccee.2009.25.