Cryptarithmetic

Cryptarithm, mathematical recreation in which the goal is to decipher an arithmetic problem in which letters have been substituted for numerical digits.
The term *crypt-arithmetic* was introduced in 1931, when the following multiplication problem appeared in the Belgian journal *Sphinx*:

```
  ABC
   DE
 ----
  FEC
 DEC
 ----
 HGBC
```

Cryptarithm now denotes mathematical problems usually calling for addition, subtraction, multiplication, or division and replacement of the digits by letters of the alphabet or some other symbols.[1]

The problem discussed here involves a simple addition of two terms consisting of characters resulting in a result term. The goal is to map each character to a digit such that the summation of two numeric terms matches the character mapping of the result term.
Here are some of the constraints:

- Leading character for each term cannot be mapped to 0
- There should be no more than 10 distinct characters. For example, abcde+efghi=ijklm has no solutions (there are 12 distinct characters)
- The summation should be the longest word. For example, ab+cde=fg has no solutions (cde is longer than fg)
- The summation cannot be too long. For example, ab+cd=efgh has no solutions (ab and cd range from 10 to 99, so the summation cannot be more than 198)[2]

For example,

```
   SEND  ->  9567
 + MORE  ->  1085
 ------     -----
  MONEY  -> 10652
```

Due to the nature of the problem, there might exist more than one solution to it. The goal is to find all such solutions.

One way to approach this problem is to try all the possibilities for each character and check for the correctness.

One way is to try to fill the columns from right to left, by assigning (unifying) valid digits to characters as we move to the left. At the beginning, the digit pool will have all 10 digits to choose from. Once we assign a digit to a character, we remove it from the digit pool. Doing this keeps us from picking the already assigned digit again for a different character (eliminating possibilities). When we reach to the left most column, and successfully assign a digit to the final character, we get the one possible solution to the problem.

At any point during assignment (unification), if we come across an invalid choice, we will skip to the next available choice until we exhaust all the choices from the digit pool. In which case, we backtrack to the previous level, make a different choice and continue from there. This mimics the depth-first-search which is a core functionality of Prolog. To achieve the same in Python, we will make use of PyLog.

First, we will setup the problem into a proper representation.

- Create a dictionary of a character to an uninstantiated PyValue for all unique characters
- Create 4 lists of PyValues for carry, term1, term2 and sum
- Length of each list would be the length of the sum term plus 1
- Pad all three terms with PyValue(0) and fill the remaining with uninstantiated PyValue instance from the dictionary
- Fill the carry list with unique uninstantiated PyValues
- Create a list leading_digits that maintains all the PyValue instances that are at the beginning of any term. We use this to avoid unifying digit 0 to any of these PyValue.

Here is how it will look like:

```
{
  ‘D’: </,0>,
  ‘N’: </,1>,
  ‘E’: </,2>,
  ‘S’: </,3>,
  ‘R’: </,4>,
  ‘O’: </,5>,
  ‘M’: </,6>,
  ‘Y’: </,7>
}
```

Lists

</,14>	</,13>	</,12>	</,11>	</,10>	<0,9>	<-	CARRY
<0,8>	<0,8>	</,3>	</,2>	</,1>	</,0>	<-	TERM1
<0,8>	<0,8>	</,6>	</,5>	</,4>	</,2>	<-	TERM2
<0,8>	</,6>	</,5>	</,1>	</,2>	</,7>	<-	SUM

Here, each <value, obj_id> represents a PyValue object. As discussed above, each list contains a PyValue object. It can be seen that a character is represented by a same PyValue object across all 3 arrays (i.e. Term1, Term2 and Sum array). For example, ‘E’ appears in all 3 terms, and is represented by same object </,2> in all 3 lists.

Some of the fields are grayed out. These are the leading characters, and therefore cannot be assigned to a digit 0. We keep track of this by looking at leading_digits list.

Here is a code that achieves the above setup:

```
def set_up_puzzle(t1: str, t2: str, sum: str, _Z: PyValue) -> \
                  Optional[Tuple[List[PyValue], List[PyValue], List[PyValue],
List[PyValue], List[PyValue]]]:
  """
  Convert the initial string representation to (uninstantiated) PyValues.
  t1 and t2 are the numbers to be added. sum is the sum.
  _Z is PyValue(0). It will be replaced by leading blanks.
  """
  Var_Letters = sorted(list(set(t1 + t2 + sum)))
  if len(Var_Letters) > 10:
    print(f'Too many variables: {Var_Letters}')
    return
  Vars_Dict = {V: PyValue() for V in Var_Letters}
  length = len(sum) + 1
  T1 = (length - len(t1)) * [_Z] + letters_to_vars(t1, Vars_Dict)
  T2 = (length - len(t2)) * [_Z] + letters_to_vars(t2, Vars_Dict)
  Sum = (length - len(sum)) * [_Z] + letters_to_vars(sum, Vars_Dict)
  # Leading_Digits are the variables that should not be assigned 0.
  Leading_Digits = letters_to_vars({t1[0], t2[0], sum[0]}, Vars_Dict)
  Carries = [PyValue() for _ in range(length - 1)] + [PyValue(0)]
  return (Carries, T1, T2, Sum, Leading_Digits)
```

Now that we have a setup, we can start solving the puzzle. In order to solve the puzzle, all we need is the parameters Carries, T1, T2, Sum and Leading_Digits created above. If it finds a valid solution, it yields at that point. We capture this yield, which acts like a halt in the execution of a function and prints out the state of all the PyValue objects to the console. This is basically one solution to the puzzle. We can iterate through this generator function until exhaustion to find out all possible solutions to the puzzle. Here is a code that iterates through the solve() generator function.

```
def solve_crypto(t1: str, t2: str, sum: str):
  _Z = PyValue(0)
  (Carries, T1, T2, Sum, Leading_Digits) = set_up_puzzle(t1, t2, sum, _Z)
  want_more = None
  Blank = PyValue(' ')
  for _ in solve(Carries, T1, T2, Sum, Leading_Digits):
    # We have a solution.
    # Replace the leading _Z zeros with blanks and convert each number to a string.
    # We can discard T1[0], T2[0], and Sum[0] because we know they will be 0.
    (t1_out, t2_out, tot_out) = (solution_to_string(T, _Z, Blank) for T in [T1[1:],
T2[1:], Sum[1:]])
```

```
    print()
    print(f'  {t1}  -> {t1_out}')
    print(f'+ {t2}  -> {t2_out}')
    print(f'{"-" * (len(sum)+1)}     {"-" * len(sum)}')
    print(f' {sum}  -> {tot_out}')
    ans = input('\nLook for more solutions? (y/n) > ').lower( )
    want_more = ans[0] if len(ans) > 0 else 'n'
    if want_more != 'y':
      break
  if want_more == 'y':
    print('No more solutions.')
```

Let's talk about how the solve() function solves the puzzle. It basically starts filling columns from right to left. fill_column() takes PyValue objects of a column and all the available digits to choose from. Once it completes filling a column, it yields back the remaining digits in the form of list.

```
def solve(Carries: List[PyValue],
          Term1: List[PyValue],
          Term2: List[PyValue],
          Sum: List[PyValue],
          Leading_Digits: List[PyValue]):
  """
  Solve the problem.
  The two embedded functions below refer to the lists in solve's params.
  The lists never change, but their elements are unified with values.
  No point is copying the lists repeatedly. So embed the functions that refer to them.
  """

  def fill_column(PVs: List[PyValue], index: int, digits_in: List[int]):
    """
    PVs are the digits in the current column to be added together, one from each term.
    digits-in are the digits that have not yet been assigned to a Var.
    Find digits in digits_in that make the column add up properly.
    Return (through yield) the digits that are not yet used after the new assignments.
    We do this recursively on PVs--even though we are currently assuming only two
terms.
    """
    if not PVs:
      # We have instantiated the digits to be added.
      # Instantiate Sum_Dig (if possible) and Carries[index - 1] to the total.
      # Completing the column is a bit more work than it might seem.
      (carry_in, digit_1, digit_2) = (D.get_py_value( ) for D in [Carries[index],
Term1[index], Term2[index]])
      total = sum([carry_in, digit_1, digit_2])
      (carry_out, sum_dig) = divmod(total, 10)
      yield from complete_column(carry_out, Carries[index-1], sum_dig, Sum[index],
digits_in, Leading_Digits)

    else:
      # Get head and tail of PVs.
      [PV, *PVs] = PVs
      # If PV already has a value, nothing to do. Go on to the remaining PVs.
      if PV.is_instantiated( ):
        yield from fill_column(PVs, index, digits_in)
      else:
        # Give PV one of the available digits. Through "backup" all digits will be
tried.
        for i in range(len(digits_in)):
          if not (digits_in[i] == 0 and PV in Leading_Digits):
            for _ in unify(PV, digits_in[i]):
              yield from fill_column(PVs, index, digits_in[:i] + digits_in[i + 1:])

  def solve_aux(index: int, digits_in: List[int]):
    """ Traditional addition: work from right to left. """
    # When we reach 0, we're done.
    if index == 0:
      # Can't allow a carry to this position.
      if Carries[0].get_py_value() == 0:
        yield
      else:
        # If we reach index == 0 but have a carry into the last column, fail.
        # Won't have such a carry with only two terms. But it might happen with many
terms,.
        return
    else:
      for digits_out in fill_column([Term1[index], Term2[index]], index, digits_in):
          yield from solve_aux(index-1, digits_out)

  yield from solve_aux(len(Carries)-1, list(range(10)))




def complete_column(carry_out: int, Carry_Out_Dig: PyValue,
                    sum_dig: int, Sum_Dig: PyValue,
                    digits_in: List[int], Leading_Digits):
  """
  If Sum_Dig (the variable representing the digit in the sum for this column) is not
yet instantiated,
  instantiate it to sum_dig  (if that digit is available). If Sum_Dig is already
instantiated, ensure
  it is consistent with the sum_dig. Instantiate Carry_Out_Dig to carry_out.
  """
  # Is Sum_Dig uninstantiated? If so, instantiate it to sum_digit if possible.
  # Then instantiate Carry_Out_Dig, and return (yield) digits_in with sum_digit
removed.
  if not Sum_Dig.is_instantiated():
    if sum_dig not in digits_in:
      # sum_dig is not available in digits_in. Fail, i.e., return instead of yield.
      return

    # sum_dig is available in digits_in. Give it to Sum_Dig as long as this does not
give
    # 0 to one of the leading digits.
    if not (sum_dig == 0 and Sum_Dig in Leading_Digits):
      for _ in unify_pairs([(Carry_Out_Dig, carry_out), (Sum_Dig, sum_dig)]):
        # Remove sum_digit from digits_in
        i = digits_in.index(sum_dig)
        yield digits_in[:i] + digits_in[i + 1:]

  # If Sum_Dig is instantiated, is it equal to sum_digit?
  # If so, instantiate Carry_Out_Dig and return the current digits_in.
  elif sum_dig == Sum_Dig.get_py_value( ):
    for _ in unify(Carry_Out_Dig, carry_out):
      yield digits_in
```

[1]: https://www.britannica.com/science/cryptarithm
[2]: http://bach.istc.kobe-u.ac.jp/llp/crypt.html