Cryptarithmetic Puzzles

Cryptarithmetic puzzles, also known as word math puzzles, are a type of logic puzzle that involves converting letters into digits to solve a mathematical equation. Here's a breakdown of their key aspects:

Objective:

- The main objective of a Cryptarithmetic puzzle is to decipher the hidden digit assignment for each letter, such that the resulting equation holds true when performing the mathematical operations (addition, subtraction, multiplication, etc.) on the digit equivalents of the letters.

Example:

SEND + MORE = MONEY

In this example:

- Each unique letter represents a unique digit (0-9).
- The goal is to find the digit assignment for each letter (S, E, N, D, M, O, R, Y) such that the sum of SEND + MORE equals MONEY.

Constraints:

- **Unique digit assignment:** Each letter can only represent one unique digit between 0 and 9. There can't be duplicate digit assignments for different letters.
- **No leading zeros:** The first digit of any resulting number (represented by a word) cannot be 0. This avoids confusion and ensures a valid mathematical representation.
- **Valid mathematical operation:** The digits assigned to the letters must satisfy the given mathematical operation (addition, subtraction, etc.) for the equation to hold true.

Formulating the Search Problem

We can formulate the word-value puzzle as a search problem with the following components:

- ➢ **State Space:** The state space consists of all possible letter-digit mappings for the unique letters in the words. Each state is a dictionary where keys are unique letters and values are digits between 0 and 9.
- ➢ **Initial State:** The initial state is an empty dictionary, representing no letters assigned to any digits yet.
- ➢ **Operators:** The operators are actions that assign a digit to an unassigned letter. An operator takes a current state (letter-digit mapping) and a letter-digit pair as input and returns a new state where the letter in the pair is assigned the corresponding digit.
- ➢ **Goal Test:** The goal test function checks if a given state (letter-digit mapping) is a valid solution. It uses validate_solution to verify that there are no leading zeros, all digits are unique, and the word-value relation holds for all words.

Search Tree:

Both DFS and BFS explore the state space using a search tree (implicitly in BFS) or by explicitly building the tree (optional for DFS). Each node in the tree represents a state (letter-digit mapping). The root node is the initial state. The children of a node are generated by applying all possible operators (assigning unassigned letters with available digits) to the current state. The search continues until a goal state (valid solution) is found or all possibilities are exhausted.

Observations on Algorithms

- **Heuristic in BFS:**

The heuristic used in the BFS algorithm is to sort the remaining letters based on the number of available digits for each letter. This heuristic aims to prioritize letters that have fewer available digits left, as these are likely to lead to a quicker solution. By exploring these letters first, the algorithm can potentially find a solution faster.

- **Pruning in BFS:**

The pruning strategy in BFS involves discarding branches of the search tree that are guaranteed to lead to invalid solutions. This is achieved by checking the constraints of the puzzle at each step and only continuing with the exploration if the current partial solution is still valid. If a partial solution violates any constraint, the algorithm prunes that branch and does not explore it further. This pruning helps reduce the search space and improve the efficiency of the algorithm.

- **Heuristic in DFS:**

In the DFS algorithm, a heuristic is used to prioritize assigning digits to letters based on their frequency in the words. This heuristic aims to start with letters that occur more frequently, as these are likely to have a more significant impact on the overall solution. By exploring these letters first, the algorithm can potentially reduce the search space and find a solution faster.

Best Algorithm Choice:

The best algorithm choice depends on the specific characteristics of the word-value puzzle:

- For smaller puzzles with a high likelihood of a solution near the root of the search tree, DFS might be faster due to its focused exploration.
- For larger puzzles with many potential invalid branches, BFS might be more efficient as it prioritizes pruning these branches early.
- If finding the shortest solution (minimum number of digit assignments) is crucial, BFS is guaranteed to find it.

Code Explanation:

Functions:

time_function (decorator): This decorator measures the execution time of a function and prints it after the function call.

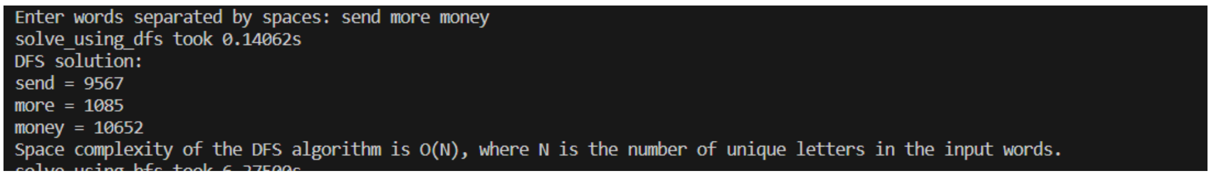
validate_solution (function): This function takes the list of words and a letter-digit mapping as input. It calculates the sum of the digit values for each word and checks if the sum of all words except the last one is equal to the sum of the last word. If the condition holds, it returns the letter-digit mapping as a valid solution; otherwise, it returns None.

validate_partial_solution (function): This function checks two constraints for a partial letter-digit mapping:

- No leading zeros: It verifies that no word starts with a letter mapped to 0.
- Unique digit assignment: It ensures that no digit is assigned to multiple letters. If any constraint is violated, it returns False; otherwise, it returns True.

solve_using_dfs (function): This function implements the DFS approach to solve the puzzle:

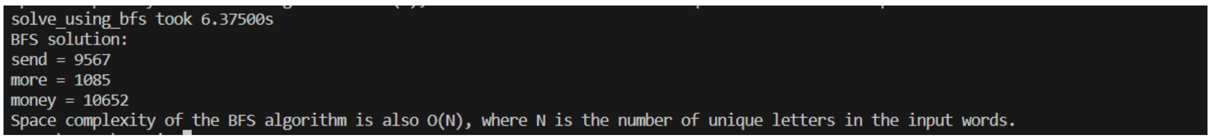
- It identifies the unique letters in all words.
- It creates a list of available digits (0-9).
- It iterates over the unique letters in descending order of their frequency (most frequent first). This prioritizes assigning digits to letters appearing in more words.
- For each letter, it sorts the available digits in descending order of frequency (if the digit appears in any remaining words). This prioritizes assigning larger digits to letters with higher potential value.
- It uses itertools.permutations to generate all possible permutations of digits for the unique letters.
- It checks for leading zeros and skips invalid permutations.
- It uses validate_solution to check if the current permutation is a valid solution.
- If a valid solution is found, it returns the letter-digit mapping.
    - Output:



```
Enter words separated by spaces: send more money
solve_using_dfs took 0.14062s
DFS solution:
send = 9567
more = 1085
money = 10652
Space complexity of the DFS algorithm is O(N), where N is the number of unique letters in the input words.
```

bfs_heuristic(function): This function implements the BFS approach to solve the puzzle:

- Similar to DFS, it identifies unique letters and available digits.
- It uses a queue (deque) to store partial letter-digit mappings.
- It starts by adding all possible permutations (without leading zeros) of available digits to the queue.
- In each iteration, it dequeues the first mapping from the queue.
- It identifies the remaining unassigned letters based on the current mapping.
- It sorts the remaining letters in descending order based on the number of available digits that can be assigned to them (prioritizing assigning digits to the most constrained letters).
- It iterates through the available digits and checks if the digit can be assigned to the current letter without violating constraints.
- It creates a new mapping with the assigned digit and adds it to the queue.
- It prunes the search by skipping branches with invalid partial solutions using validate_partial_solution.
- If a valid solution is found, it returns the letter-digit mapping.
- Output:



```
solve_using_bfs took 6.37500s
BFS solution:
send = 9567
more = 1085
money = 10652
Space complexity of the BFS algorithm is also O(N), where N is the number of unique letters in the input words.
```